# Concrete and Abstract Cost Semantics for Spreadsheets

Bock, Alexander Asp; Bøgholm, Thomas; Sestoft, Peter; Thomsen, Bent; Thomsen, Lone Leth

*Publication date:*
2018

*Document Version*
Også kaldet Forlagets PDF


# Concrete and Abstract Cost Semantics for Spreadsheets

**Alexander Asp Bock**
**Thomas Bøgholm**
**Peter Sestoft**
**Bent Thomsen**
**Lone Leth Thomsen**

**IT University Technical Report Series** **TR-2018-203**

**ISSN 1600–6100** **December 2018**

**ISSN 1600–6100**

**ISBN 978-87-7949-369-8**

**Copies may be obtained by contacting:**

**IT University of Copenhagen**
**Rued Langgaards Vej 7**
**DK-2300 Copenhagen S**
**Denmark**

**Telephone:** **+45 72 18 50 00**
**Telefax:** **+45 72 18 50 01**
**Web** **`www.itu.dk`**

# Concrete and Abstract Cost Semantics for Spreadsheets

Alexander Asp Bock
Thomas Bøgholm
Peter Sestoft
Bent Thomsen
Lone Leth Thomsen

**Abstract**

We give a simple but precise operational semantics for the evaluation of extended spreadsheet formulas, with array formulas, sheet-defined functions and closures, as found in the Funcalc spreadsheet platform [1]. We build on this to give a simple cost semantics, inspired by [2], for evaluation of a spreadsheet formula and for full and minimal recalculation of a spreadsheet. Following the ideas presented by Schmidt [3] we provide a big-step trace-based abstract interpretation for the cost semantics. We then present a set of functions which can be used to calculate the cost of executing an evaluation of a spreadsheet expression following Gomez et al. [4], inspired by Rosendahl [5]. These functions are related to the above operational semantics, cost semantics and abstract interpretation.

The above semantic presentations all form the formal foundations for various cost calculations implemented in the Funcalc spreadsheet platform. These calculations are evaluated experimentally.

## 1 Introduction

Every day spreadsheets are used by millions of people, ranging from pupils doing their school hand-ins to complex financial, medical or scientific computations. In 2017 it was estimated that there were 13-25 million spreadsheet developers worldwide [6], i.e. people developing complex computations using spreadsheets. Yet, despite their widespread use the semantics of spreadsheet computations is rather underdeveloped and it is almost impossible to analyze the computational cost of spreadsheet computations. This paper takes its outset in the semantics for simple spreadsheets sketched in section 1.8 of the book Spreadsheet Implementation Technology [1].

In this paper, we give a simple but precise operational semantics for the evaluation of extended spreadsheet formulas, with array formulas, sheet-defined

functions and closures, as found in the Funcalc spreadsheet platform [1].

We build on this semantic definition to give a simple cost semantics, inspired by [2], for evaluation of a spreadsheet formula for full and minimal recalculation of a spreadsheet.

We follow the ideas presented by Schmidt [3] and provide a big step trace-based abstract interpretation for the cost semantics.

We then present a set of functions which can be used to calculate the cost of executing an evaluation of a spreadsheet expression following Gomez et al. [4], inspired by Rosendahl [5]. These functions are related to the above operational semantics, cost semantics and abstract interpretation.

The above semantic presentations all form the formal foundations for various cost calculations implemented in the Funcalc spreadsheet platform. These calculations are evaluated experimentally.

The calculation of execution cost is paramount to investigations of various approaches to parallelizing the execution of spreadsheet programs pursued in the Popular Parallel Programming (P3) project[1], e.g. as used in [7].

The rest of this paper is organized as follows: The evaluation semantics for simple Funcalc expressions is elaborated in Section 2 and semantics for extended spreadsheet expressions is developed in Section 3. In section crefsec-cost-semantics a precise cost semantics is built on top of the extended semantics. This semantics is presented in Section 5 and in Section 6. This cost semantics serve as a foundation for implementations described in Section 7 and in Section 10. The extended evaluation semantics for Funcalc is extended to compute with unknown values in Section 8, which serves as a first step towards an approximate cost analysis described in Section 9. In Section 11, we present results pertaining to the execution time and precision of the various cost analyses that were implemented as described in Section 10. Finally, we present conclusions and future work in Section 12.

## 2 Simple Spreadsheet Semantics

This section describes the evaluation of simple spreadsheets without array formulas and sheet-defined functions, and hence without array values and closures. It is reproduced from parts of [1, Section 1.8] and included here for background; readers familiar with the subject may skip to Section 3.

The simplified formulas used in this section are described in Figure 1. One simplification is to represent a constant cell `n` by a constant formula `=n`, although most spreadsheet programs would distinguish them. Another simplification is to leave out cell area expressions $ca_1 : ca_2$; these will be introduced in Section 3.

[1] https://www.itu.dk/~sestoft/p3/

| | | | |
|---|---|---|---|
| $e$ | ::= | `n` | number constant |
| | \| | `ca` | cell reference |
| | \| | `IF`($e_1$,$e_2$,$e_3$) | conditional expression |
| | \| | `RAND`() | volatile function |
| | \| | `F`($e_1,\ldots,e_n$) | built-in function call |

**Figure 1:** Syntax of the simplified formula language.

To describe the evaluation of such formulas, we use the semantic sets and functions defined in Figure 2. These are sometimes called semantic domains, but here they are ordinary sets and partial functions. For instance, $Value = Number + Error$ is the set of values, where a value $v$ is either a proper number such as 0.42 in set $Number$ or an error such as `#DIV/0!` in set $Error$. The set $Addr$ contains cell addresses $ca$ such as B2. For presentational simplicity, some additional error values (such as `#NAME!`) and additional kinds of values (such as strings), found in realistic spreadsheet programs, have been left out. They are easily added to the semantics studied here.

To describe the formulas of a worksheet, we use a map $\phi : Addr \to Expr$ so that when $ca \in Addr$ is a cell address, $\phi(ca)$ is the formula in cell $ca$. If cell $ca$ is blank, then $\phi(ca)$ is undefined. The domain of $\phi$ is $dom(\phi) = \{\ ca \mid \phi(ca) \text{ is defined } \}$, the set of cell addresses that have a formula, that is, the set of non-blank cells. The $\phi$ function is not affected by recalculation, only by editing the sheet.

The result of a recalculation is modelled by function $\sigma : Addr \to Value$, where $\sigma(ca)$ is the computed value in cell $ca$. The $\sigma$ function gets updated by each recalculation (see Section 2.2).

| | | | | |
|---|---|---|---|---|
| $n$ | $\in$ | $Number$ | $=$ | { proper numbers } |
| | | $Error$ | $=$ | { `#DIV/0!`, `#CYCLE!` } |
| $ca$ | $\in$ | $Addr$ | $=$ | { cell addresses } |
| $v$ | $\in$ | $Value$ | $=$ | $Number + Error$ |
| $e$ | $\in$ | $Expr$ | $=$ | { formulas, see Figure 1 } |
| $\phi$ | | | $\in$ | $Addr \to Expr$ |
| $\sigma$ | | | $\in$ | $Addr \to Value$ |

**Figure 2:** Sets and maps used in the spreadsheet semantics: $Number$ is the set of proper floating-point numbers, excluding NaNs and infinities; $Error$ is the set of error values; $Addr$ the set of cell addresses; $Value$ the set of values (either number or error); and $Expr$ the set of formulas.

## 2.1 Semantics of Formula Evaluation

The semantics for formulas is given as a natural semantics [8], a variant of operational semantics [9], using inference rules that involve big-step evaluation judgments. An evaluation judgment has the form $\sigma \vdash e \Downarrow v$, which says: When $\sigma$ describes the calculated values of all cells, then formula $e$ may evaluate to value $v$. Note that $v$ may be a number value or an error value.

To understand inference rules, consider this rule:

$$\frac{\sigma \vdash e_i \Downarrow v_i \in \mathit{Error}}{\sigma \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow v_i} \; (e5e)$$

This inference rule consists of a premise above the line and a conclusion below the line. The conclusion concerns the value of a function call expression $\mathtt{F}(e_1, \ldots, e_n)$, and the premise concerns the value of one of the call's argument expressions $e_i$. The rule can be read as follows: If there is some argument expression $e_i$ that may evaluate to an error value $v_i$, then the function call may evaluate to the error value $v_i$ also. That is, the rule describes the propagation of errors from argument to result in a function call. If multiple arguments $e_i$ and $e_j$ may evaluate to different error values $v_i$ and $v_j$, then the rule does not specify which error will be propagated to the call's result.

For another example, consider this rule, also for a function call $\mathtt{F}(e_1, \ldots, e_n)$ with $n$ arguments:

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \notin \mathit{Error} \quad \ldots \quad \sigma \vdash e_n \Downarrow v_n \notin \mathit{Error}}{\sigma \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)} \; (e5v)$$

This rule has $n$ premises and can be read as follows: If all argument expressions $e_1, \ldots, e_n$ may evaluate to non-error values $v_1, \ldots, v_n$, then the value of the function call is obtained by applying the actual function $f$ to these values, as in $f(v_1, \ldots, v_n)$.

The "may" is important because, in general, an expression may evaluate to multiple different values. For instance, `RAND()` may evaluate to any number between 0.0 (included) and 1.0 (excluded). Hence, `7+1/RAND()` may evaluate to some number greater than 7+1 or to the error `#DIV/0!` in case `RAND()` produces 0.0.

The complete set of inference rules that describe when a formula evaluation judgment $\sigma \vdash e \Downarrow v$ holds is given in Figure 3. Note that there are five groups of rules (e1), (e2x), (e3x), (4), (e5x), each corresponding to one of the five kinds of formulas in Figure 1. Also, the formula fragments that appear in the premises are always smaller than the formula that appears in the conclusion. Hence, one can make a conclusion about a given formula through a finite number of rule applications.

$$\frac{}{\sigma \vdash \mathtt{n} \Downarrow n}\ (e1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma \vdash \mathtt{ca} \Downarrow 0.0}\ (e2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma \vdash \mathtt{ca} \Downarrow v}\ (e2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \in Error}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v_1}\ (e3e)$$

$$\frac{\sigma \vdash e_1 \Downarrow 0.0 \qquad \sigma \vdash e_3 \Downarrow v}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v}\ (e3f)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad v_1 \neq 0.0 \qquad \sigma \vdash e_2 \Downarrow v}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v}\ (e3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma \vdash \mathtt{RAND}() \Downarrow v}\ (e4)$$

$$\frac{\sigma \vdash e_i \Downarrow v_i \in Error}{\sigma \vdash \mathtt{F}(e_1,\ldots,e_n) \Downarrow v_i}\ (e5e)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \notin Error \qquad \ldots \qquad \sigma \vdash e_n \Downarrow v_n \notin Error}{\sigma \vdash \mathtt{F}(e_1,\ldots,e_n) \Downarrow f(v_1,\ldots,v_n)}\ (e5v)$$

**Figure 3:** Evaluation rules for simplified spreadsheet formulas.

The formula evaluation rules in Figure 3 may be explained as follows:

- Rule (e1) says that a number constant `n` evaluates to that constant's value.
- Rule (e2b) says that a reference $ca$ to a blank cell, that is, one for which $\sigma(ca)$ is not defined, gives value 0.0.
- Rule (e2v) says that a reference $ca$ to a non-blank cell evaluates to the value $\sigma(ca)$ calculated for that cell. This value may be a number or an error.
- Rule (e3e) says that the expression $\mathtt{IF}(e_1,e_2,e_3)$ may evaluate to error $v_1$ if the condition $e_1$ may evaluate to error $v_1$.

- Rule (e3f) says that `IF`($e_1$,$e_2$,$e_3$) may evaluate to value $v$ provided the condition $e_1$ may evaluate to the non-error number zero and the "false branch" $e_3$ may evaluate to $v$.
- Rule (e3t) says that `IF`($e_1$,$e_2$,$e_3$) may evaluate to value $v$ provided the condition $e_1$ may evaluate to some non-error non-zero number $v_1$ and the "true branch" $e_2$ may evaluate to $v$.
- Rule (e4) says that function call `RAND`() may evaluate to any (non-error) number $v$ greater than or equal to zero and less than one. Hence, this rule models nondeterministic choice. It permits a formula involving `RAND()` to produce a different result on each evaluation. However, it does not *require* `RAND()` to produce a different number every time it is called. Such a requirement would not make sense; by definition, a random number generator is permitted to return whatever result it wants. So according to this operational semantics, `RAND()` might consistently return 0.42 whenever it is called, although that would be rather disappointing and useless.
- Rule (e5e) says that a call `F`($e_1, \ldots, e_n$) to a built-in function `F` may evaluate to error $v_i$ if one of its arguments $e_i$ may evaluate to error $v_i$. Note that if more than one argument may evaluate to an error, then the function call may evaluate to any of these. Hence, the semantics does not prescribe an evaluation order for arguments, such as a left to right or right to left.
- Rule (e5v) says that a call `F`($e_1, \ldots, e_n$) to a function `F` may evaluate to value $v$ if each argument $e_i$ may evaluate to non-error value $v_i$, and applying the actual function $f$ to arguments $(v_1, \ldots, v_n)$ produces value $v$. The final result $v$ may be a number such as 5, for instance, if the call is $+(2,3)$; or it may be an error such as `#DIV/0!`, for instance, if the call is `/(1.0, 0.0)`.

## 2.2 Semantics of Simple Recalculation

Now that we know how to evaluate a formula, given values of all cells in the worksheet, we can describe the semantics of a recalculation. A recalculation must find a value for every non-blank cell $ca$ in the sheet, and that value $\sigma(ca)$ must agree with the formula $\phi(ca)$ held in that cell. These are the central consistency requirements on a recalculation, formally described in Figure 4. These requirements leave it completely unspecified how the recalculation works, whether it recalculates all or only some cells, whether it does so sequentially or in parallel, whether it guesses the values or computes them, and so on. This underspecification is essential to permit a range of implementation strategies and optimizations.

(1) $dom(\sigma) = dom(\phi)$
(2) $\forall ca \in dom(\phi).\ \sigma \vdash \phi(ca) \Downarrow \sigma(ca)$

**Figure 4:** The consistency requirements on recalculation for simple formulas. Requirement (1) says that a recalculation must find a value $\sigma(ca)$, possibly an error, for every non-blank cell $ca$. Requirement (2) says that the computed value $\sigma(ca)$ must agree with the cell's formula $\phi(ca)$.

# 3 Funcalc Semantics

In this section we extend the simple spreadsheet semantics from Section 2. We first extend the expressions and semantic sets to account for array formulas and to account for sheet-defined functions. We then discuss the modelling of array formulas, which is slightly more general than strictly necessary. With array formulas in the expression language, we need to extend the semantics for ordinary sheets. This turns out to be a smooth extension where "old" rules just pass around the additional semantic environment for array expressions. We then extend the semantics to account for function sheets and round of the section with a discussion of the rules for calling sheet-defined

functions, as these rules are some of the more unusual aspects of this semantics.

## 3.1 Extended Expressions and Semantic Sets

The simple spreadsheet semantics from Section 2 must be expanded in two orthogonal directions: to account for array formulas and to account for sheet-defined functions. This requires extension to the formula expression language, shown in Figure 5, and to the set of values and semantic maps, shown in Figure 6.

A cell area reference $ca_1 : ca_2$ refers to a block of cells spanned by the two opposing "corner" cells $ca_1$ and $ca_2$. In Funcalc, a cell area reference can refer to an ordinary sheet only, not to a function sheet.

An array formula is here modelled as an underlying formula $ae$ which is itself just an expression, expected to evaluate to an array value, that is, an array of values. That array value's components are distributed over a target cell area, with one such component in each cell. This is explained in more detail in Section 3.2.

We model a closure as a partial application, that is, a named sheet-defined function $sdf$ with a prefix $[u_1, \ldots, u_k]$ of its argument values given, where $0 \leq k \leq arity(sdf)$; see Figure 6. A closure is created by `CLOSURE` from a sheet-defined function $sdf$ by giving it values for some or all of its arguments. A partially applied closure $e_0$ may be given further arguments, as in currying, also using `CLOSURE`. An `APPLY` call of a closure $e_0$ must provide all the remaining $n$

| | | | |
|---|---|---|---|
| $e$ | ::= | $\mathtt{n}$ | number constant |
| | \| | $\mathtt{ca}$ | cell reference |
| | \| | $\mathtt{IF}(e_1,e_2,e_3)$ | conditional expression |
| | \| | $\mathtt{RAND}()$ | volatile function |
| | \| | $\mathtt{F}(e_1,\ldots,e_n)$ | call to built-in function |
| | \| | $\mathtt{ca_1} : \mathtt{ca_2}$ | cell area reference |
| | \| | $ae[i,j]$ | array formula component |
| | \| | $sdf(e_1,\ldots,e_n)$ | call to sheet-defined function |
| | \| | $\mathtt{CLOSURE}(sdf,e_1,\ldots,e_k)$ | closure creation |
| | \| | $\mathtt{CLOSURE}(e_0,e_1,\ldots,e_n)$ | closure partial application |
| | \| | $\mathtt{APPLY}(e_0,e_1,\ldots,e_n)$ | closure full application |
| $ae$ | ::= | $e$ | array expression |

**Figure 5:** Syntax of the Funcalc extended formula language, with five additional syntactic constructs: a cell area reference, an access to component $(i, j)$ of an array formula $ae$, a call of a sheet-defined function, creation of a closure from a sheet-defined function $sdf$, and call of a closure $e_0$.

arguments, where $k + n = arity(sdf)$, and will call the underlying sheet-defined function.

In the simple semantics for formula evaluation on ordinary sheets, the recalculation consistency requirements could be stated in terms of the formula $\phi(ca)$ in a given cell and its post-recalculation value $\sigma(ca)$.

To account for array formulas, we need the post-recalculation value $\alpha(ae)$ of each underlying (presumably array-valued) expression $ae$. This underlying value will be shared by all the array formula's components, see rule (e7) in Figure 10. In Funcalc, array formulas are allowed on ordinary sheets only, not on function sheets.

To further account for a call to a sheet-defined function, we need the value $\rho(ca)$ of the function sheet cells $ca$ used during the call of the function. Each call, also each recursive call, has its own fresh $\rho$ map, and the map is ephemeral: there is no way to refer to a function sheet cell value after the function has returned. Hence $\rho$ is similar to a stack frame in ordinary programming language implementation.

Note that $\alpha$ is not needed when evaluating a sheet-defined function, because function sheets cannot contain array formulas. Also, $\rho$ is not needed when evaluating cells on an ordinary sheet, because there is no way to refer to a function sheet cell value after the function has returned. The revised post-recalculation consistency requirements are shown in Figure 7.

$$
\begin{array}{llll}
n & \in & Number & = & \{ \text{ proper numbers } \} \\
av & \in & ArrVal & = & \{ (w, h, [[v_{ij} \mid i \leq w, j \leq h]]) \} \\
fv & \in & FunVal & = & \{ (sdf, [u_1, \ldots, u_k]) \} \\
 & & Error & = & \{ \texttt{\#DIV/0!}, \texttt{\#CYCLE!} \} \\
ca & \in & Addr & = & \{ \text{ cell addresses } \} \\
v, u & \in & Value & = & Number + Error + ArrVal + FunVal \\
e & \in & Expr & = & \{ \text{ formulas, see Figure 5 } \} \\
\phi & & & \in & Addr \to Expr \\
\sigma & & & \in & Addr \to Value \\
\alpha & & & \in & Expr \to Value \\
\rho & & & \in & Addr \to Value
\end{array}
$$

**Figure 6:** Sets and maps used in the Funcalc extended spreadsheet semantics. There are the following differences relative to Figure 2: $av \in ArrVal$ is a one based array value with $w \times h$ component values $v_{ij}$ and $fv \in FunVal$ is a function value (closure) consisting of a function name $sdf$ and $0 \leq k \leq arity(sdf)$ given argument values $u_i$. In this extended semantics, $v \in Value$ is either a number or error or array value or function value. Array values are needed because of cell area expressions $ca_1 : ca_2$, and function values because of `CLOSURE` expressions. There are new semantic maps: $\alpha$ maps an array expression $ae$ to its value, and $\rho$ maps a function sheet cell address to its value.

$$
\begin{array}{ll}
(1) & dom(\sigma) = dom(\phi) \\
(2) & \forall ca \in dom(\phi).\ \sigma, \alpha \vdash \phi(ca) \Downarrow \sigma(ca) \\
(3) & \forall ae \in dom(\alpha).\ \sigma, \alpha \vdash ae \Downarrow \alpha(ae)
\end{array}
$$

**Figure 7:** The consistency requirements on recalculation with array formulas and sheet-defined functions. The requirement (2) is extended with $\alpha$ to account for array formulas. The new requirement (3) says that a recalculation must find a single value $\alpha(ae)$ for each array expression $ae$ underlying an array formula; this value will be used in all components of the array formula via applications of (2).

## 3.2 Modelling Array Formulas

An array formula is an expression, such as `transpose(e2:g3)`, whose result is an array value and where the components of this array value are spread over a target cell area, such as B2:C4. This situation is shown in Figure 8 where the target cell area has been marked and the formula has been written into cell B2; the array formula is then created by an incantation such as pressing Ctrl+Shift+Enter in Excel. The effect of doing so is shown in Figure 9 where the target cells B2:C4 contain the transpose of the values in E2:G3.

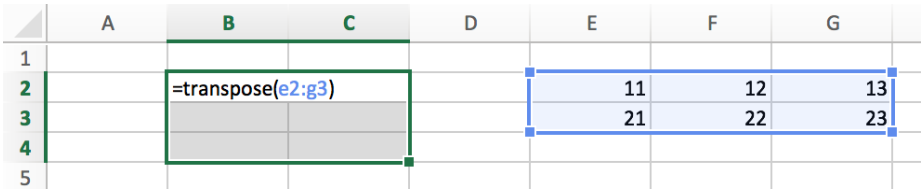


**Figure 8:** Entering an array formula with target cell area B2:C4.

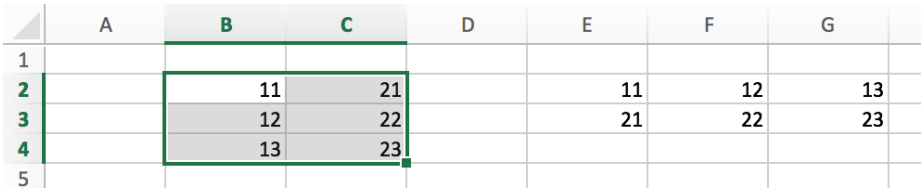


**Figure 9:** The six cells in target cell area B2:C4 each contain one component of the result of the underlying array expression `transpose(e2:g3)`.

Editing any cell in the range E2:G3 would cause the array expression to be recalculated and the values in B2:C4 to be updated. The underlying array expression is evaluated at most once in each recalculation.

In Funcalc extended spreadsheet expressions, we model the individual cells belonging to an array formula by the syntax $ae[i, j]$ that suggests indexing into the value of the underlying array expression $ae$. In the index $(i, j)$ the $i$ and $j$ are constants, with $i$ ranging over columns and $j$ over rows, both one-based. For instance, cell B2 in Figures 8 and 9 would contain the expression $ae[1, 1]$ where $ae$ is the underlying array expression `transpose(e2:g3)`, cell B3 would contain $ae[1, 2]$, cell C2 would contain $ae[2, 1]$, and so on. Indexing into an error value produces that error value itself, so we need no separate "error version" of rule (e7) in Figure 10.

The syntax in Figure 5 allows an array formula component $ae[i, j]$ to appear anywhere an expression can, also nested inside another expression. This is overly general, since $ae[i, j]$ need appear only at top level in a cell formula, not in nested expressions. We could enforce this restriction by introducing an

additional syntactic category of *cell* which can be an expression $e$ or an array formula component $ae[i, j]$, and remove the latter from the syntactic category of expressions. This would also be in better agreement with the implementation of Funcalc. However, since the excess generality is harmless, and getting rid of it would lead to additional largely administrative rules without semantic significance, we stick to the simple but slightly too permissive syntax in Figure 5.

### 3.3 Extended Semantics for Ordinary Sheets

An evaluation judgment in the extended semantics for ordinary formula evaluation has the form $\sigma, \alpha \vdash e \Downarrow v$. It says that when $\sigma$ describes the calculated values of all cells and $\alpha$ describes the values of all array expressions underlying array formulas, then formula $e$ may evaluate to value $v$, where $v$ may be a number value, an error value, an array value or a closure value.

The rules defining the judgment $\sigma, \alpha \vdash e \Downarrow v$ are shown in Figure 10. They are a smooth extension of those in Figure 3. The "old" rules (e1) through (e5v) have been extended to pass around also the $\alpha$ array expression map. The six new rules (e6), (e7), (e8), (e9), (e10), and (e11) account for cell area references, array formulas, calls to sheet-defined functions, closure creation, and closure calls. They correspond exactly to the six new syntactic constructs in Figure 5.

$$\frac{}{\sigma, \alpha \vdash \texttt{n} \Downarrow n} \; (e1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma, \alpha \vdash \texttt{ca} \Downarrow 0.0} \; (e2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma, \alpha \vdash \texttt{ca} \Downarrow v} \; (e2v)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \in Error}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v_1} \; (e3e)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow 0.0 \qquad \sigma, \alpha \vdash e_3 \Downarrow v}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v} \; (e3f)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \qquad v_1 \neq 0.0 \qquad \sigma, \alpha \vdash e_2 \Downarrow v}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v} \; (e3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma, \alpha \vdash \texttt{RAND}() \Downarrow v} \; (e4)$$

$$\frac{\sigma, \alpha \vdash e_i \Downarrow v_i \in \mathit{Error}}{\sigma, \alpha \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow v_i} \; (e5e)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1 \notin \mathit{Error} \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n \notin \mathit{Error}}{\sigma, \alpha \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)} \; (e5v)$$

$$\frac{\begin{array}{c}(c_1, r_1) = ca_1 \quad (c_2, r_2) = ca_2 \quad (c_l, c_r) = \mathit{sort}(c_1, c_2) \quad (r_t, r_b) = \mathit{sort}(r_1, r_2) \\ w = c_r - c_l + 1 \quad h = r_b - r_t + 1\end{array}}{\sigma, \alpha \vdash \mathtt{ca_1 : ca_2} \Downarrow \mathit{ArrVal}(w, h, [[\sigma[c_l + i, r_t + j] \mid i \leq w, j \leq h]])} \; (e6)$$

$$\frac{}{\sigma, \alpha \vdash ae[i, j] \Downarrow \alpha(ae)[i, j]} \; (e7)$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_1 \Downarrow v_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n \\ \mathit{def}(\mathit{sdf}) = (\mathit{out}, [\mathit{in}_1, \ldots, \mathit{in}_n], \mathit{cells}) \\ \rho'(\mathit{in}_1) = v_1 \quad \ldots \quad \rho'(\mathit{in}_n) = v_n \\ \forall \mathit{ca} \in \mathit{dom}(\rho') \setminus \{\mathit{in}_1, \ldots, \mathit{in}_n\}.\ \rho', \sigma \vdash \phi(\mathit{ca}) \Downarrow \rho'(\mathit{ca})\end{array}}{\sigma, \alpha \vdash \mathit{sdf}(e_1, \ldots, e_n) \Downarrow \rho'(\mathit{out})} \; (e8)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow u_1 \quad \ldots \quad \sigma, \alpha \vdash e_k \Downarrow u_k}{\sigma, \alpha \vdash \mathtt{CLOSURE}(\mathit{sdf}, e_1, \ldots, e_k) \Downarrow \mathit{FunVal}(\mathit{sdf}, [u_1, \ldots, u_k])} \; (e9)$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_0 \Downarrow \mathit{FunVal}(\mathit{sdf}, [u_1, \ldots, u_k]) \\ \sigma, \alpha \vdash e_1 \Downarrow v_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n\end{array}}{\sigma, \alpha \vdash \mathtt{CLOSURE}(e_0, e_1, \ldots, e_n) \Downarrow \mathit{FunVal}(\mathit{sdf}, [u_1, \ldots, u_k, v_1, \ldots, v_n])} \; (e10)$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_0 \Downarrow \mathit{FunVal}(\mathit{sdf}, [u_1, \ldots, u_k]) \\ \sigma, \alpha \vdash e_1 \Downarrow v_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n \\ \mathit{def}(\mathit{sdf}) = (\mathit{out}, [\mathit{in}_1, \ldots, \mathit{in}_{k+n}], \mathit{cells}) \\ \rho'(\mathit{in}_1) = u_1 \ \ldots \ \rho'(\mathit{in}_k) = u_k \quad \rho'(\mathit{in}_{k+1}) = v_1 \ \ldots \ \rho'(\mathit{in}_{k+n}) = v_n \\ \forall \mathit{ca} \in \mathit{dom}(\rho') \setminus \{\mathit{in}_1, \ldots, \mathit{in}_{k+n}\}.\ \rho', \sigma \vdash \phi(\mathit{ca}) \Downarrow \rho'(\mathit{ca})\end{array}}{\sigma, \alpha \vdash \mathtt{APPLY}(e_0, e_1, \ldots, e_n) \Downarrow \rho'(\mathit{out})} \; (e11)$$

**Figure 10:** Evaluation rules for Funcalc extended spreadsheet formulas.

In Figure 10, the new rule (e6) says that a cell area reference $ca_1 : ca_2$ evaluates to an array value $\mathit{ArrVal}(w, h, [[v_{ij}]])$ with $w$ columns, $h$ rows, and $w \cdot h$ values $v_{ij}$ obtained from the cell value map $\sigma$. The utility function $\mathit{sort}(x, y)$ returns

the pair of the least and greatest of $x$ and $y$, so that $c_l$ and $c_r$ are the leftmost and rightmost column indices, and $r_t$ and $r_b$ are the top and bottom row indices, of the cell area $ca_1 : ca_2$. This is necessary because it is legal to enter a cell area reference such as B1:A2, thereby giving the cell area's upper right (B1) and lower left (A2) corners, and that should be "normalized" to A1:B2 which gives the cell area's upper left and lower right corners instead, for proper calculation of width and height. One cannot just forbid the B1:A2 notation because it arises naturally by copying a mixed relative-absolute cell area reference such as A1:$A$1.

Rule (e7) says that component $(i, j)$ of an array formula is found by looking up the value $av = \alpha(ae)$ of the underlying array expression and then indexing into that value by $av[i, j]$, where such indexing must produce an error value if $av$ is not an array value or does not have a component $(i, j)$. Note that in the judgment left-hand side, the indexing notation is syntactic, and in the right-hand side it is semantic.

Rule (e8) describes how to call a sheet-defined function *sdf* that has output cell address *out*, that has input cell addresses $[in_1, \ldots, in_n]$, and that is defined using only cells at addresses *cells* (excluding the input cells but including the output cell) on a separate function sheet. We assume that the cells defining *sdf* are given by $def(sdf) = (out, [in_1, \ldots, in_{k+n}], cells)$, and that all these cells are in $dom(\phi)$ — that is, $\phi$ describes also the formulas of the function sheet on which *sdf* is defined.

The evaluation of a call to a sheet-defined function *sdf* proceeds as follows. First evaluate the call's argument expressions to values $v_1, \ldots, v_n$, then postulate a fresh environment $\rho'$ in which the called function's input cells $[in_1, \ldots, in_n]$ have these values and all other cells used by the function have consistent values. Then the function call's value is the value $\rho'(out)$ of the function's output cell. Judgments of the form $\rho', \sigma \vdash e \Downarrow v$ are defined in Figure 11 below. For a discussion of the function call semantics, and an example, see Section 3.5.

It is natural to expect the new $\rho'$ environment to be defined for all the sheet-defined function's cells, as in $dom(\rho') = \{in_1, \ldots, in_n\} \cup cells$, but actually it suffices for $dom(\rho')$ to be the set of cells needed to compute the value of the output cell *out*. See also the discussion in Section 5.1.

The expression language is call-by-value, and a call to a sheet-defined function is strict in the sense that every argument is evaluated before the function is called, regardless of whether the function's body actually refers to the argument's value.

A call to a sheet-defined function is not error-strict: the function is called even though some argument $e_i$ evaluates to an error value. Hence the argument evaluation premises are simpler than in rule (e5v) for calling built-in functions, and there is no error-case rule (e8e) corresponding to rule (e5e). Naturally, the same holds for constructing or calling a closure in rules (e9), (e10) and (e11).

Rule (e9) says that to evaluate a closure creation, evaluate the $k$ given argu-

ments and create a function value consisting of the function name *sdf* and the $k$ resulting values, whether errors or proper values.

Rule (e10) says that to further apply a partially applied closure, evaluate $e_0$ to a closure containing $k$ already given arguments, then evaluate the $n$ further arguments, and create a new closure containing the $k+n \leq arity(sdf)$ argument values given so far.

Rule (e11) says that to evaluate a call to closure $FunVal(sdf, [u_1, \ldots, u_k])$, evaluate the $n$ given arguments, then proceed to call the sheet-defined function *sdf* on its $k + n$ arguments in the same manner as described in rule (e8).

## 3.4 Extended Semantics for Function Sheets

An evaluation judgment in the semantics for sheet-defined functions has the form $\rho, \sigma \vdash e \Downarrow v$. It says that when $\rho$ describes the calculated values of all cells on the function sheet defining the function and $\sigma$ describes the calculated values of all cells on ordinary sheets, then formula $e$ may evaluate to value $v$, where $v$ may be a number value, an error value, an array value or a closure value.

$$\frac{}{\rho, \sigma \vdash \texttt{n} \Downarrow n} \; (f1)$$

$$\frac{ca \notin dom(\rho) \qquad ca \notin dom(\sigma)}{\rho, \sigma \vdash \texttt{ca} \Downarrow 0.0} \; (f2b)$$

$$\frac{ca \in dom(\rho) \qquad \rho(ca) = v}{\rho, \sigma \vdash \texttt{ca} \Downarrow v} \; (f2f)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\rho, \sigma \vdash \texttt{ca} \Downarrow v} \; (f2v)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow v_1 \in Error}{\rho, \sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v_1} \; (f3e)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow 0.0 \qquad \rho, \sigma \vdash e_3 \Downarrow v}{\rho, \sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v} \; (f3f)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow v_1 \qquad v_1 \neq 0.0 \qquad \rho, \sigma \vdash e_2 \Downarrow v}{\rho, \sigma \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v} \; (f3t)$$

$$\frac{0.0 \leq v < 1.0}{\rho, \sigma \vdash \texttt{RAND}() \Downarrow v} \; (f4)$$

$$\frac{\rho, \sigma \vdash e_i \Downarrow v_i \in Error}{\rho, \sigma \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow v_i} \; (f5e)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow v_1 \notin Error \quad \ldots \quad \rho, \sigma \vdash e_n \Downarrow v_n \notin Error}{\rho, \sigma \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)} \; (f5v)$$

$$\frac{\begin{array}{c} ca_1 \in dom(\sigma) \quad ca_2 \in dom(\sigma) \\ (c_1, r_1) = ca_1 \quad (c_2, r_2) = ca_2 \quad (c_l, c_r) = sort(c_1, c_2) \quad (r_t, r_b) = sort(r_1, r_2) \\ w = c_r - c_l + 1 \quad h = r_b - r_t + 1 \end{array}}{\rho, \sigma \vdash \texttt{ca}_1 : \texttt{ca}_2 \Downarrow ArrVal(w, h, [[\sigma[c_l + i, r_t + j] \mid i \leq w, j \leq h]])} \; (f6)$$

$$\frac{\begin{array}{c} \rho, \sigma \vdash e_1 \Downarrow v_1 \quad \ldots \quad \rho, \sigma \vdash e_n \Downarrow v_n \\ def(sdf) = (out, [in_1, \ldots, in_n], cells) \\ \rho'(in_1) = v_1 \quad \ldots \quad \rho'(in_n) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_n\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca) \end{array}}{\rho, \sigma \vdash sdf(e_1, \ldots, e_n) \Downarrow \rho'(out)} \; (f8)$$

$$\frac{\rho, \sigma \vdash e_1 \Downarrow u_1 \quad \ldots \quad \rho, \sigma \vdash e_k \Downarrow u_k}{\rho, \sigma \vdash \texttt{CLOSURE}(sdf, e_1, \ldots, e_k) \Downarrow FunVal(sdf, [u_1, \ldots, u_k])} \; (f9)$$

$$\frac{\begin{array}{c} \rho, \sigma \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]) \\ \rho, \sigma \vdash e_1 \Downarrow v_1 \quad \ldots \quad \rho, \sigma \vdash e_n \Downarrow v_n \end{array}}{\rho, \sigma \vdash \texttt{CLOSURE}(e_0, e_1, \ldots, e_n) \Downarrow FunVal(sdf, [u_1, \ldots, u_k, v_1, \ldots, v_n])} \; (f10)$$

$$\frac{\begin{array}{c} \rho, \sigma \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]) \\ \rho, \sigma \vdash e_1 \Downarrow v_1 \quad \ldots \quad \rho, \sigma \vdash e_n \Downarrow v_n \\ def(sdf) = (out, [in_1, \ldots, in_{k+n}], cells) \\ \rho'(in_1) = u_1 \ \ldots \ \rho'(in_k) = u_k \quad \rho'(in_{k+1}) = v_1 \ \ldots \ \rho'(in_{k+n}) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca) \end{array}}{\rho, \sigma \vdash \texttt{APPLY}(e_0, e_1, \ldots, e_n) \Downarrow \rho'(out)} \; (f11)$$

**Figure 11:** Evaluation rules for Funcalc sheet-defined functions.

The rules defining the judgment $\rho, \sigma \vdash e \Downarrow v$ are shown in Figure 11. They are intentionally very similar to those for evaluation of ordinary (extended)

spreadsheet formulas shown in Figure 10 — after all, the whole point of sheet-defined functions is that they should be familiar to spreadsheet users.

However, there is an additional rule (f2f) for lookup of a cell address on a function sheet; rule (f6) requires a cell area reference to refer to an ordinary worksheet, not a function sheet; and there is no rule (f7) for array formulas, which are not allowed in function sheets.

## 3.5 Discussion: Calling a Sheet-Defined Function

The rules (e8) and (f8) for calls to sheet-defined functions, and the corresponding closure call rules (e11) and (f11), are some of the more unusual aspects of this semantics. The core idea in these rules is that a fresh environment $\rho'$ is postulated for evaluation of the called function *sdf*. Informally, this corresponds to (A) the creation of a fresh copy of the function sheet on which *sdf* is defined, and also to (B) the creation of a new stack frame to hold the function's arguments and local variables. Explanation (A) is what a Funcalc spreadsheet user should have in mind, and (B) is what the Funcalc implementation actually does. Without loss of generality we can assume that each sheet-defined function is defined in its own sheet, and only the cells used in the definition of *sdf* need be recalculated.

There is nothing mysterious or unusual about the "freshness" of $\rho'$. Formally, $\rho'$ is no different from $v$ in rule (e4): it is just a variable representing some value (here an environment) that must satisfy the premises.

In explanation (A), the fresh sheet copy $\rho'$ is used as follows: fill the input cells $[in_1, \ldots, in_n]$ with the values of the evaluated arguments; recalculate the sheet as usual for spreadsheets; return the output cell's value as the result of the call; and discard the sheet copy. These steps are faithfully reflected in rules (e8) and (f8), with the "recalculate as usual" step expressed by the last premise

$$\forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_n\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca)$$

This premise is meant to reflect the standard spreadsheet consistency requirement (2) in Figures 4 and 7, but for the temporary function sheet's cell values $\rho'$ instead of an ordinary sheet's cell values $\sigma$.

Consider the simple sheet-defined function `F` in Figure 12, with input cells B2 and B3, intermediate cell B4 containing the formula `=IF(RAND()<0.5, B2, B3)` and output cell B5 containing the formula `=B4+B4`.

The set *cells* of cells making up the function's body is $\{\text{B4}, \text{B5}\}$. To evaluate a call `F(1,5)` to this function, the semantics will create a fresh environment $\rho'$ that must have $\rho'(\text{B2}) = 1$ and $\rho'(\text{B3}) = 5$. Now we can additionally have either $\rho'(\text{B4}) = 1$ and so $\rho'(\text{B5}) = 2$, or $\rho'(\text{B4}) = 5$ and so $\rho'(\text{B5}) = 10$; these are the only two possibilities according to the semantics. Hence the call `F(1,5)` must

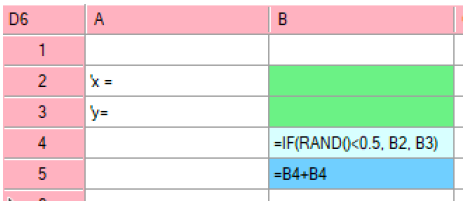


**Figure 12:** Sheet-defined function `F` with input cells B2 and B3, output cell B5, and an intermediate cell B4.

return 2 or 10. It cannot return $1 + 5 = 6$ because both occurrences of B4 in `=B4+B4` must have the same value $\rho'(\text{B4})$.

Note that the universal quantification $\forall ca \in dom(\rho') \ldots$ in the last premise of rules (e8), (f8), (e11) and (f11) ranges over a finite set: the cells used to define function $sdf$. Hence in any concrete application of these rules, the quantifier gives rise to only a finite number of proof subtrees.

A call from a sheet-defined function to another one, or indeed a recursive call to the function itself, is handled naturally by rules (f8) and (f11) through postulating a new fresh environment $\rho'$ for the called function, distinct from the calling function's $\rho$. In terms of explanation (A) given above, a fresh copy of the defining function sheet is created for each recursive call; and in terms of (B), a fresh stack frame is allocated for each recursive call. Also, all these sheet copies, or stack frames, coexist until the function calls return. (However, as a semantics-preserving optimization, the actual Funcalc implementation may deallocate the old stack frame early in case of a tail call).

Infinite recursion in a sheet-defined function is reflected in the operational semantics by an attempt to build an infinitely deep derivation tree, through an infinite number of applications of rules (f8) or (f11). Obviously this is not possible, so no value can be derived for an infinite recursive call, not even an error value. Note that this is different from the meaning of a cyclic dependency in an ordinary spreadsheet, for which an error value could be derived (by the semantics and the implementation): just put $\sigma(ca) = \texttt{\#CYCLE!} \in Error$ for all the cells $ca$ cyclically dependent on each other.

# 4 Cost Semantics

In this section we extend the evaluation semantics to a cost semantics, which in addition to a possible computed value of the expression describes the possible cost of computing it. More precisely, the semantics describes the *work*, that is, uni-processor cost [2], of the computation. In a parallel implementation, some of that work may be performed in parallel.

This section gives a cost semantics for simple spreadsheet expressions by building on the Section 2 evaluation semantics, then Section 5 gives a cost semantics for Funcalc extended spreadsheet expressions by building on the Section 3 evaluation semantics.

In all cases the amount of work is described by a non-negative integer in $Nat_0$ representing some notion of computation step, for instance the number of evaluation rule applications, plus some measure of the cost of calling a built-in function (such as `SUM` over a range of cells). This notion of work can reasonably be assumed to be within a constant factor of the actual number of nanoseconds required to evaluate an expression.

## 4.1 Cost Semantics for Simple Formulas

For simplicity we start by giving a cost semantics for simple spreadsheet formulas as described in Section 2. The evaluation judgment $\sigma \vdash e \Downarrow v$ gets extended to $\sigma \vdash e \Downarrow v, c$ where $v$ is a computed value of the expression $e$ and $c$ is the cost of computing that value. This judgment states that when $\sigma$ describes the calculated values of all cells, then formula $e$ may evaluate to value $v$ at computational cost $c$. As in Section 2.1, the semantics is nondeterministic ("may") in the sense that the evaluation of an expression $e$ could produce many different values $v$ at many different costs $c$. See also the discussion at the end of Section 4.2.

It is quite straightforward to extend the evaluation semantics rules in Figure 3 to the new cost semantics rules given in Figure 13.

$$\frac{}{\sigma \vdash \mathtt{n} \Downarrow n, 1}\ (c1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma \vdash \mathtt{ca} \Downarrow 0.0, 1}\ (c2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma \vdash \mathtt{ca} \Downarrow v, 1}\ (c2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \in Error}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v_1, 1 + c_1}\ (c3e)$$

$$\frac{\sigma \vdash e_1 \Downarrow 0.0, c_1 \qquad \sigma \vdash e_3 \Downarrow v, c_3}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_3}\ (c3f)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \neq 0.0 \qquad \sigma \vdash e_2 \Downarrow v, c_2}{\sigma \vdash \mathtt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_2}\ (c3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma \vdash \texttt{RAND}() \Downarrow v, 1} \; (c4)$$

$$\frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\ \sigma \vdash e_j \Downarrow v_j, c_j \qquad v_i \in \mathit{Error} \text{ for some } i \in J \end{array}}{\sigma \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow v_i, 1 + \sum_{j \in J} c_j} \; (c5e)$$

$$\frac{\begin{array}{c} \sigma \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma \vdash e_n \Downarrow v_n, c_n \\ \forall i.\ v_i \notin \mathit{Error} \end{array}}{\sigma \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n), 1 + \sum_{j=1,n} c_j + \mathit{work}(f, v_1, \ldots, v_n)} \; (c5v)$$

**Figure 13:** Cost (or work) semantics rules for simplified spreadsheet formulas.

The formula evaluation rules in Figure 13 may be explained as follows:

- Rule (c1) says that evaluating a number constant `n` requires 1 computation step, and similarly for cell references by rules (c2b) and (c2v).
- Rule (c3e) says that if $e_1$ may evaluate to error $v_1$ in $c_1$ computation steps, then $\texttt{IF}(e_1,e_2,e_3)$ may evaluate to error $v_1$ in $1 + c_1$ computation steps.
- Rule (c3f) says that if $e_1$ may evaluate to the non-error number 0.0 in $c_1$ computation steps and the "false branch" $e_3$ may evaluate to $v$ in $c_3$ computation steps, then $\texttt{IF}(e_1,e_2,e_3)$ may evaluate to value $v$ in $1+c_1+c_3$ computation steps.
- Rule (c3t) is similar, for when $e_1$ may evaluate to some non-error non-zero number $v_1$ in $c_1$ computation steps.
- Rule (c4) says that function call `RAND()` may evaluate to any (non-error) number $v$ greater than or equal to zero and less than one, in one computation step.
- Rule (c5e) is quite different from the corresponding evaluation rule (e5e) in Figure 3. It says that an implementation may choose to evaluate just a subset $\{e_j \mid j \in J\}$ of the arguments when some $e_i$ with $i \in J$ evaluates to an error $v_i$, and then let $v_i$ be the result of the function call. Also, it says that the total cost of this is the cost $\sum_{j \in J} c_j$ of evaluating that subset of arguments, plus one. The rationale for this is discussed in Section 4.2.
- Rule (c5v) says that if each argument $e_i$ may evaluate to a non-error value $v_i$ in $c_i$ computation steps and applying the actual function $f$ to argument values $(v_1, \ldots, v_n)$ produces value $v$ at a cost of $\mathit{work}(f, v_1, \ldots, v_n)$ computation steps, then the call $\texttt{F}(e_1, \ldots, e_n)$ may evaluate to value $v$ using a total of $1 + \sum_{j=1,n} c_j + \mathit{work}(f, v_1, \ldots, v_n)$ computation steps.

  Here $\mathit{work}(f, v_1, \ldots, v_n)$ describes the cost of applying function $f$ to argument values $(v_1, \ldots, v_n)$. For instance, one would expect $\mathit{work}(+, v_1, v_2) =$

1 since the cost of addition is independent of the numbers added. By contrast, for functions on array values one would expect the cost to depend on the argument array size; for instance, $work(transpose, v_1) = w \cdot h$ when array value $v_1$ has $w$ columns and $h$ rows.

Making each cost rule add 1 to the cost incurred by subexpression evaluations may appear very simplistic. It means that the cost semantics essentially counts the number of rule applications. A more realistic cost semantics might replace each occurrence of "1" with a suitable constant indicating a number of nanoseconds for the operation, such as 1 for evaluating a constant, 8 for evaluating a cell reference, 40 for a call to `RAND`, and similar. However, if we are interested in cost up to a constant factor, counting the number of rule applications works just as well, and avoids some notational clutter. Also, the real time cost of something as simple as a cell reference may vary from 1 ns to 80 ns depending on whether the relevant data is already in cache or not.

## 4.2 Rationale for Cost of an Error Argument

While most of the cost semantics rules in Figure 13 are obvious extensions of the evaluation rules in Figure 3, this is not the case for rule (c5e) which is quite different from rule (e5e). Here we explain why.

It is possible to imagine a cost rule (c5bad) similar to rule (e5e), like this:

$$\frac{\sigma \vdash e_i \Downarrow v_i, c_i \qquad v_i \in Error}{\sigma \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow v_i, 1 + c_i} \; (c5bad)$$

This rule says that if one of the arguments $e_i$ may evaluate to an error $v_i$ using $c_i$ computation steps, then the call $\mathtt{F}(e_1, \ldots, e_n)$ to a function `F` may evaluate to error $v_i$ in $1 + c_i$ computation steps. However, this cost is unrealistically low: a conforming implementation would have to correctly guess which (if any) argument expression $e_i$ can evaluate to an error, and then evaluate only that expression. Such an implementation would seem implausibly clever.

A more realistic rule might stipulate instead that the cost is the sum of the costs of evaluating all argument expressions. This corresponds to implementations that would evaluate all arguments before checking whether any of them evaluates to an error. However, this is needlessly pessimistic since an implementation may stop evaluating arguments once one of them evaluates to an error.

Another realistic cost rule might correspond to implementations that evaluate argument expressions $e_1, e_2, \ldots$ from left to right until one of them (if any) evaluates to an error. However, this restricts the possible implementations and would preclude or complicate parallel evaluation of arguments.

Instead we propose rule (c5e) in Figure 13 which corresponds to implementations that may evaluate the argument expressions in any order (or in parallel) but may avoid evaluating all of them in case one evaluates to an error. As shown in

the rule this corresponds to choosing a subset $J \subseteq \{1, \ldots, n\}$ of the argument indices and evaluating only the $e_j$ for which $j \in J$, to values $v_j$ at costs $c_j$, where one of the $v_j$ is an error, and then stating that the total cost of the call is the sum $\sum_{j \in J} c_j$ of the costs of the arguments actually evaluated, plus one.

Since the set $J$ may be chosen in many ways, this rule introduces nondeterminism in the evaluation cost, in addition to nondeterminism in the computed value. Note also that rule (c5e) encompasses all three alternative rules discussed above, by choosing $J = \{i\}$ as the singleton set for which $v_i$ is an error (using unrealistically perfect foresight), or $J = \{1, \ldots, n\}$ to evaluate all arguments, or $J = \{1, \ldots, i\}$ as the least prefix of argument indices for which $v_i$ is an error.

## 4.3 Cost of Simple Recalculation

Sections 4.1 and 4.2 above gave evaluation-and-cost rules for evaluation of spreadsheet formulas. How do we describe the cost of a full recalculation or minimal recalculation in terms of these?

First, we introduce a cost environment $\gamma : Addr \to Nat_0$ such that $\gamma(ca)$ is the cost of evaluating the formula at cell address $ca$. Then we slightly change the recalculation consistency requirements to also record the cost of evaluation for each cell, as shown on Figure 14.

$$\begin{array}{ll} (1) & dom(\sigma) = dom(\gamma) = dom(\phi) \\ (2) & \forall ca \in dom(\phi).\ \sigma \vdash \phi(ca) \Downarrow \sigma(ca), \gamma(ca) \end{array}$$

**Figure 14:** Recalculation consistency requirements recording also evaluation cost, for simple formulas. The judgment $\sigma \vdash e \Downarrow v, c$ is defined in Figure 13. Compared to the consistency requirements in Figure 4, requirement (2) has been extended to record the evaluation cost of cell $ca$ in $\gamma(ca)$.

Using the cost environment $\gamma$ we can now express the cost of a full recalculation of a spreadsheet described by $\phi$. This is simply the cost of evaluating the formula of every non-blank cell once:

$$fullcost \quad = \quad \sum_{ca \in dom(\phi)} \gamma(ca)$$

Likewise we can express the cost of a minimal recalculation initiated by editing a single cell $ca_0$ in a previously consistent spreadsheet. Let the consistent spreadsheet be represented by $\phi$ and $\sigma$. Let $dirty(ca_0)$ be the transitive closure under the "supports" relation (also called the "dependents" relation) of the set containing cell $ca_0$ and every cell whose formula is volatile. That is, $dirty(ca_0)$ is the set of cells that need to be recalculated when $ca_0$ has changed. Then build new environments $\sigma'$ and $\gamma'$ such that

(1) $dom(\sigma') = dom(\gamma') = dom(\phi)$
(2) $\forall ca \in dom(\phi).\ \sigma' \vdash \phi(ca) \Downarrow \sigma'(ca), \gamma'(ca)$
(3) $\forall ca \notin dirty(ca_0).\ \sigma'(ca) = \sigma(ca)$

That is, $\sigma'$ agrees with $\sigma$ on all cells that have not been recalculated, but may have new values for those cells that have been recalculated. Now the total cost of a minimal recalculation after a change to cell $ca_0$ is the sum of the costs of evaluating the cells in $dirty(ca_0)$:

$$minimalcost \quad = \quad \sum_{ca \in dirty(ca_0)} \gamma'(ca)$$

Since $dirty(ca_0)$ is defined via the "supports" relation it may be an overapproximation of the set of cells that really need to be evaluated in a minimal recalculation. How best to express the actual cost of a minimal recalculation without prescribing an evaluation mechanism is not completely clear, but a nondeterministic or underdetermined approach similar to that in Section 4.2 may be feasible. Whether this is worth the effort and complexity is not obvious.

# 5 Cost Semantics for Extended Formulas

In this section we extend the cost semantics to cover array formulas and sheet-defined functions. The cost semantics for Funcalc extended spreadsheet formulas is given by judgments of the form $\sigma, \alpha \vdash e \Downarrow v, c$ which say that when $\sigma$ describes cell values and $\alpha$ describes array expression values, expression $e$ may evaluate to value $v$ at a cost of $c$ computation steps. The rules defining these judgments are given in Figure 15.

$$\frac{}{\sigma, \alpha \vdash \texttt{n} \Downarrow n, 1}\ (g1)$$

$$\frac{ca \notin dom(\sigma)}{\sigma, \alpha \vdash \texttt{ca} \Downarrow 0.0, 1}\ (g2b)$$

$$\frac{ca \in dom(\sigma) \qquad \sigma(ca) = v}{\sigma, \alpha \vdash \texttt{ca} \Downarrow v, 1}\ (g2v)$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \in Error}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v_1, 1 + c_1}\ (g3e)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow 0.0, c_1 \qquad \sigma, \alpha \vdash e_3 \Downarrow v, c_3}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_3}\ (g3f)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad v_1 \neq 0.0 \qquad \sigma, \alpha \vdash e_2 \Downarrow v, c_2}{\sigma, \alpha \vdash \texttt{IF}(e_1,e_2,e_3) \Downarrow v, 1 + c_1 + c_2} \; (g3t)$$

$$\frac{0.0 \leq v < 1.0}{\sigma, \alpha \vdash \texttt{RAND}() \Downarrow v, 1} \; (g4)$$

$$\frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\ \sigma, \alpha \vdash e_j \Downarrow v_j, c_j \qquad v_i \in \mathit{Error} \text{ for some } i \in J \end{array}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots,e_n) \Downarrow v_i, 1 + \sum_{j \in J} c_j} \; (g5e)$$

$$\frac{\begin{array}{c} \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ \forall i.\ v_i \notin \mathit{Error} \end{array}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots,e_n) \Downarrow f(v_1, \ldots, v_n), 1 + \sum_{j=1,n} c_j + \mathit{work}(f, v_1, \ldots, v_n)} \; (g5v)$$

$$\frac{\begin{array}{c} (c_1, r_1) = ca_1 \qquad (c_2, r_2) = ca_2 \qquad (c_l, c_r) = \mathit{sort}(c_1, c_2) \qquad (r_t, r_b) = \mathit{sort}(r_1, r_2) \\ w = c_r - c_l + 1 \qquad h = r_b - r_t + 1 \end{array}}{\sigma, \alpha \vdash \texttt{ca}_1 : \texttt{ca}_2 \Downarrow \mathit{ArrVal}(w, h, [[\sigma[c_l + i, r_t + j] \mid i \leq w, j \leq h]]), w \cdot h} \; (g6)$$

$$\frac{}{\sigma, \alpha \vdash ae[i, j] \Downarrow \alpha(ae)[i, j], 1} \; (g7)$$

$$\frac{\begin{array}{c} \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ \mathit{def}(\mathit{sdf}) = (\mathit{out}, [\mathit{in}_1, \ldots, \mathit{in}_n], \mathit{cells}) \\ \rho'(\mathit{in}_1) = v_1 \qquad \ldots \qquad \rho'(\mathit{in}_n) = v_n \\ \forall ca \in \mathit{dom}(\rho') \setminus \{\mathit{in}_1, \ldots, \mathit{in}_n\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca) \end{array}}{\sigma, \alpha \vdash \mathit{sdf}(e_1, \ldots,e_n) \Downarrow \rho'(\mathit{out}), 1 + \sum_{j=1,n} c_j + \sum_{ca \in \mathit{dom}(\gamma')} \gamma'(ca)} \; (g8)$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow u_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_k \Downarrow u_k, c_k}{\sigma, \alpha \vdash \texttt{CLOSURE}(\mathit{sdf},e_1, \ldots,e_k) \Downarrow \mathit{FunVal}(\mathit{sdf}, [u_1, \ldots, u_k]), 1 + \sum_{j=1,k} c_j} \; (g9)$$

$$\frac{\begin{array}{c} \sigma, \alpha \vdash e_0 \Downarrow \mathit{FunVal}(\mathit{sdf}, [u_1, \ldots, u_k]), c_0 \\ \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \end{array}}{\sigma, \alpha \vdash \texttt{CLOSURE}(e_0,e_1, \ldots,e_n) \Downarrow \mathit{FunVal}(\mathit{sdf}, [u_1, \ldots, u_k, v_1, \ldots, v_n]), 1 + c_0 + \sum_{j=1,n} c_j} \; (g10)$$

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_0 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_0 \\ \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ def(sdf) = (out, [in_1, \ldots, in_{k+n}], cells) \\ \rho'(in_1) = u_1 \ \ldots \ \rho'(in_k) = u_k \quad \rho'(in_{k+1}) = v_1 \ \ldots \ \rho'(in_{k+n}) = v_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca)\end{array}}{\sigma, \alpha \vdash \texttt{APPLY}(e_0,e_1,\ldots,e_n) \Downarrow \rho'(out), 1 + c_0 + \sum_{j=1,n} c_j + \sum_{ca \in dom(\gamma')} \gamma'(ca)} \ (g11)$$

**Figure 15:** Cost (or work) semantics rules for Funcalc extended spreadsheet formulas. The consistency requirements on recalculation are in Figure 19.

The extended cost semantics rules in Figure 15 draw on the extended evaluation rules in Figure 10 as well as the simple cost semantics rules in Figure 13.

Rules (g1) through (g5v) are very similar to the simple cost semantics rules (c1) through (c5v). This includes the somewhat complicated case (g5e) of a function argument evaluating to an error, explained in Section 4.2.

Rule (g6) states that the cost of evaluating a cell area expression that produces an array value of $w$ columns and $h$ rows is $w \cdot h$, the number of components in the resulting array value.

Rule (g7) states that the cost of evaluating a cell that is part of an array formula is 1. This is because we require the array formula's shared underlying array expression to be evaluated at most once in a recalculation, so evaluating the cell is just a matter of indexing into the resulting array.

Rule (g8) states that the cost of calling a sheet-defined function is the cost of evaluating all arguments, plus the cost of evaluating the function body, plus one. The cost of evaluating the function body is the sum of the costs of evaluating the cells used to define the function, as described by the cost environment $\gamma'$. It is clear that $dom(\gamma')$ must equal $dom(\rho') \setminus \{in_1, \ldots, in_n\}$, but there is some flexibility in exactly which set of cells $dom(\rho')$ should be evaluated. See the discussion in Section 5.1.

Rule (g9) states that the cost of creating a closure is the cost of evaluating the $k$ given arguments, plus one.

Rule (g11) states that the cost to call a closure is the cost of evaluating the closure expression, plus the cost of evaluating the remaining arguments, plus the cost of evaluating the called function's body, plus one. Similar to rule (g8), $dom(\gamma')$ must equal $dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}$, and Section 5.1 discusses how to choose $dom(\rho')$ and hence $dom(\gamma')$.

## 5.1 The Cost of Calling a Sheet-Defined Function

The rules (g8) and (g11) for calling a sheet-defined function leave unspecified the set $dom(\rho')$ of the function's cells that should be evaluated, and hence the set

$dom(\gamma') = dom(\rho') \setminus \{in_1, \ldots, in_n\}$ whose evaluation costs should be included in the call cost.

As in rule (e8) the set $dom(\rho')$ may contain all the function's cells, but it suffices to include only those cells actually needed to compute the value of the output cell *out*. For the ordinary semantics this distinction is less important, since it does not affect the result $\rho'(out)$ of the function call, assuming that evaluation terminates. However, for the cost semantics the distinction is crucial. Obviously, evaluating cells that are not needed, and hence including them in $dom(\rho')$ and in $dom(\gamma')$ affects the cost $\sum_{ca \in dom(\gamma')} \gamma'(ca)$ of the computation.

If the value of a cell *ca* is needed, directly or indirectly, to compute the value of the output cell *out*, then *ca* must be in $dom(\rho')$. Conversely, a cell whose value is not needed by the output cell should not be in $dom(\rho')$. However, whether a cell *ca* is needed or not cannot be determined prior to evaluation. It depends both on the input cell values (the function's argument values) and on the evaluation of volatile functions such as a `RAND`. Consider the example sheet-defined function `FCT` in Figure 16.

| A10 | A | B |
|---|---|---|
| 1 | =DEFINE("fct", B5, B2, B3) | |
| 2 | 'p = | |
| 3 | 'n = | |
| 4 | | =B2*B2*B3*B3 |
| 5 | 'res = | =IF(RAND()<B2, B3, B4) |

**Figure 16:** A slightly contrived sheet-defined function `FCT` with input cells B2 and B3 and output cell B5. The formula in cell B4 needs to be evaluated only if the condition in B5 is false.

If input B2 $\geq 1$, the condition in output cell B5 is always true and the function evaluates to B3 without having to evaluate B4; and if B2 $\leq 0$, the condition in B5 is always false, and cell B4 must be evaluated to produce the result of the function.

When $0 <$ B2 $< 1$, the value of `RAND()` determines whether cell B4 really needs to be evaluated. A reasonable cost semantics should allow for leaving

out the cost of evaluating cell B4 when its value is not needed. On the other hand, it should also allow for adding in that cost, so as to correctly describe an implementation that speculatively evaluates B4 although its value may not be needed. For instance, an implementation may evaluate B4 to exploit available parallel computation resources, or simply because of the cost of unconditionally evaluating B4 is smaller than the cost of determining whether its value is needed (and then performing the relevant conditional jumps, synchronization, or the like).

Thus in the semantics there should be some freedom in choosing $dom(\rho')$ and hence $dom(\gamma')$ and hence the total cost of the function call. The choice should be subject to a consistency requirement: if cell $ca \in dom(\rho')$ and the value of $ca$ depends on cell $ca_r$, then $ca_r \in dom(\rho')$ too.

How can we describe more formally that the value of a cell $ca_r$ is needed to compute the output cell and hence the function's return value? In the Funcalc implementation, so-called evaluation conditions [1, Chapter 9] are used to control which cells must be evaluated. However, that is a particular *implementation mechanism* and should not be part of the cost semantics *specification*.

Hence we propose to specify the consistency requirement as follows. Consider an application of rule (g8) and all the inference trees that prove the judgments in the last row of premises. The domains $dom(\rho')$ and hence $dom(\gamma') = dom(\rho') \setminus \{in_1, \ldots, in_n\}$ must satisfy the following. For each $ca \in dom(\rho') \setminus \{in_1, \ldots, in_n\}$ there is an inference tree that proves

$$\rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca)$$

Now the consistency requirement says that for each function-sheet cell reference $ca_r \in cells$ encountered while building that inference tree, it is the case that $ca_r \in dom(\rho')$. In other words, any (non-input) cell $ca_r$ referred to during the evaluation of the sheet-defined function must have a value, meaning $ca_r \in dom(\rho')$, so that lookup succeeds by the cost semantics rule (h2f) for sheet-defined functions, shown in Figure 18. Also, the cost of that computation must be accounted for, meaning $ca_r \in dom(\gamma')$.

Note that this consistency requirements is loose enough to allow for speculative computation of unneeded cells, so long as this does not lead to an attempt to build an infinite inference tree, representing nonterminating recursion.

To illustrate the subtlety of the choice of whether to evaluate an unneeded cell, consider function `EX` in Figure 17. This is a slight variant of `FCT`, where crucially the trivial formula in B4 has been replaced with a recursive call `=EX(B2, B3+1)`, so that now it is essential both for termination and correct cost accounting that B4 is evaluated only when needed.

| A10 | A | B |
|---|---|---|
| 1 | =DEFINE("ex", B5, B2, B3) | |
| 2 | 'p = | |
| 3 | 'n = | |
| 4 | | =EX(B2, B3+1) |
| 5 | 'res = | =IF(RAND()<B2, B3, B4) |
| 6 | | |

**Figure 17:** A sheet-defined function `EX` such that `EX(`$p$`,1)` returns a random sample $(1, 2, \dots)$ from the geometric distribution with parameter $p$. The function definition is similar to `FCT` in Figure 16, but cell B4 contains a recursive call to `EX` itself, so now it is essential that cell B4 does not get evaluated unconditionally. By eventually evaluating B4 only when it is needed, we can achieve that a call `EX(`$p$`,`$n$`)` terminates if and only if $p > 0$.

## 5.2 Cost Semantics for Function Sheets

A cost semantics for sheet-defined functions on function sheets can be given by rules defining judgments of the form $\rho, \sigma \vdash e \Downarrow v, c$. Such judgments are referred to in rules (g8) and (g11). Because the rules are simple mixtures of the evaluation rules for function sheets in Figure 11 and the cost semantics in Figure 15, we give only one of these rules here, in Figure 18.

$$\frac{ca \in dom(\rho) \qquad \rho(ca) = v}{\rho, \sigma \vdash \mathtt{ca} \Downarrow v, 1} \; (h2f)$$

**Figure 18:** Example of cost (or work) semantics rule for Funcalc sheet-defined functions, corresponding to evaluation rule (f2f) in Figure 11. The remaining rules, which are left out here, would be similarly obvious cost versions of the other rules from that figure, except for the cost version of (f5e) as discussed in Section 4.2.

## 5.3 Cost of Extended Recalculation

The cost of recalculation for Funcalc extended formulas must account for array formulas and for sheet-defined functions.

The cost of evaluating the array expression $ae$ underlying an array formula is defined as for any other expression. We use the $\gamma$ environment also to record this cost as $\gamma(ae)$, so its type is now $\gamma : Addr + Expr \to Nat_0$. The consistency requirements for a cost semantics accounting also for array formulas are shown in Figure 19.

(1) $dom(\sigma) = dom(\phi)$
(2) $\forall ca \in dom(\phi).\ \sigma, \alpha \vdash \phi(ca) \Downarrow \sigma(ca), \gamma(ca)$
(3) $\forall ae \in dom(\alpha).\ \sigma, \alpha \vdash ae \Downarrow \alpha(ae), \gamma(ae)$
(4) $dom(\gamma) = dom(\phi) \cup dom(\alpha)$

**Figure 19:** The consistency requirements on recalculation and cost with array formulas and sheet-defined functions. The judgment $\sigma, \alpha \vdash e \Downarrow v, c$ is defined in Figure 15. Compare Figures 7 and 14.

The total cost of a full recalculation therefore is the sum of computing the formula in every cell, plus the cost of computing the array expression underlying every array formula:

$$fullcost \quad = \quad \sum_{ca \in dom(\phi)} \gamma(ca) + \sum_{ae \in dom(\alpha)} \gamma(ae)$$

We extend the $dirty(ca_0)$ set to also include array expressions that need to be recalculated (in addition to cells that need to), so now $dirty(ca_0) \subseteq Addr{+}Expr$.

Hence the cost of a minimal recalculation can be expressed as before:

$$minimalcost \quad = \quad \sum_{ca \in dirty(ca_0)} \gamma'(ca)$$

where $\gamma'$ is a cost environment determined in a similar manner as in Section 4.3.

# 6 Rules for Intrinsic Functions

In this section, we extend the operational cost semantics from Section 5 by expanding the function application rule (g5v) for a meaningful subset of intrinsic functions in Funcalc. Rule (g5v) is given below for convenience.

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \quad \ldots \quad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \qquad \forall i. v_i \notin Error}{\sigma, \alpha \vdash \mathtt{F}(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n), 1 + \sum_{j=1,n} c_j + work(f, v_1, \ldots, v_n)} \text{ (g5v)}$$

By "meaningful subset" we mean that it is not sensible or interesting to give rules for some of the intrinsic functions. For example, `EXTERN` returns the result of a call to an external library. While the returned value can be (and is) given by a plain C# `object` type, its cost is undefined. The call may perform any operation from querying a database to initiating some long-running, unknown computation that we have insufficient knowledge to approximate. Alternatively, we could give meaningful rules for some common uses for `EXTERN` such as the methods in the .NET libraries, but we forgo this here. We focus only on ordinary, interpreted sheets, as the rules for function sheets are mostly analogous.

As a starting point, consider the rule for the `SIN` function that computes the sine of its input value.

$$\frac{\sigma, \alpha \vdash \mathtt{e} \Downarrow v, c \qquad v \in Number}{\sigma, \alpha \vdash \mathtt{SIN(e)} \Downarrow sin(v), 1 + c} \text{ (sin)}$$

The rule states that if the expression $e$ may evaluate to a number $v$ at cost $c$ when $\sigma$ is an environment mapping cell addresses to values and $\alpha$ is an environment mapping array expressions to array values, then the function application expression `SIN(e)` may evaluate to the actual function application $sin(v)$ at total cost $1 + c$: 1 for the function application and $c$ for the evaluation of `e`. Similar rules can be given for `COS` and `TAN`. We use unit costs to ensure that successive application of rules to expressions are monotonically increasing. We may instead choose to define a lookup structure for mapping each intrinsic function to a cost obtained from more precise sources such as benchmarks.

We introduce a few conventions that must be borne in mind when reading the semantic rules in the following sections. We introduce a more compact notation for array values:

$$Av(w, h, [[v_{ij}]]) \triangleq ArrVal(w, h, [[v_{ij} \mid i \leq w, j \leq h]])$$

Array value indices are one-based and must be positive. When a conclusion needs to refer to an array value in a premise, we use the notation $arr = [[v_{ij}]]$. It should be clear from context what the assigned array value refers to. We mostly omit the "error rules" dealing with cases where a function argument evaluates to an error value, and ask the reader to imagine analogues of rule (g5e) in Section 5. Finally, we have deliberately left out some intrinsic functions because their expected semantics are still unclear. For full details of all functions available in Funcalc, we refer the reader to [1].

First-order intrinsic functions are given in Section 6.1, higher-order functions are given in Section 6.2.

## 6.1 Rules for First-Order Intrinsic Functions

The `NA()` function returns the special `#NA` error used to indicate that a value is not available or to indicate an unbound parameter in a partially evaluated closure. Therefore, we extend the set of errors defined in Section 2 to $Error = \{$`#DIV/0!, #CYCLE!, #NA`$\}$.

$$\frac{v \in Number}{\sigma, \alpha \vdash \texttt{NOW}() \Downarrow v, 1} \text{ (now)}$$

$$\frac{}{\sigma, \alpha \vdash \texttt{PI}() \Downarrow \pi, 1} \text{ (pi)}$$

$$\frac{}{\sigma, \alpha \vdash \texttt{NA}() \Downarrow \texttt{\#NA}, 1} \text{ (na)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \in Number}{\sigma, \alpha \vdash \texttt{ABS}(e) \Downarrow |v|, 1 + c} \text{ (abs)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \in Number}{\sigma, \alpha \vdash \texttt{ASIN}(e) \Downarrow asin(v), 1 + c} \text{ (asin)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v = 0}{\sigma, \alpha \vdash \texttt{NOT}(e) \Downarrow 1, 1 + c} \text{ (not-1)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow v, c \qquad v \neq 0}{\sigma, \alpha \vdash \texttt{NOT}(e) \Downarrow 0, 1 + c} \text{ (not-2)}$$

$$\frac{\begin{array}{cc}\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 \\ v_1 \in Number & v_2 \in Number\end{array}}{\sigma, \alpha \vdash \texttt{CEILING}(e_1, e_2) \Downarrow ceiling(v_1, v_2), 1 + c_1 + c_2} \text{ (ceiling)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2}{\sigma, \alpha \vdash e_1 = e_2 \Downarrow v_1 = v_2, 1 + c_1 + c_2} \text{ (equal)}$$

$$\frac{\begin{array}{ccc}J \subseteq \{1, \ldots, n\} & & \\ \forall j \in J.\, v_j \in Number & \forall j \in J.\, \sigma, \alpha \vdash e_j \Downarrow v_j, c_j & \exists j \in J.\, v_j = 0\end{array}}{\sigma, \alpha \vdash \texttt{AND}(e_1, \ldots, e_n) \Downarrow 0, 1 + \sum_{j \in J} c_j} \text{ (and-false)}$$

$$\frac{\begin{array}{cc}J = \{1, \ldots, n\} & \\ \forall j \in J.\, v_j \in Number \wedge v_j \neq 0 & \forall j \in J.\, \sigma, \alpha \vdash e_j \Downarrow v_j, c_j\end{array}}{\sigma, \alpha \vdash \texttt{AND}(e_1, \ldots, e_n) \Downarrow 1, 1 + \sum_{j \in J} c_j} \text{ (and-true)}$$

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n}{\sigma, \alpha \vdash \texttt{SUM}(e_1, \ldots, e_n) \Downarrow \sum_{i=1}^{n} v_i, 1 + \sum_{i=1}^{n} c_i} \text{ (sum)}$$

$$\frac{\begin{array}{ccc}\sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 & \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 \\ & v_2 \in Number \wedge v_2 \geq 0 & v_3 \in Number \wedge v_3 \geq 0\end{array}}{\sigma, \alpha \vdash \texttt{CONSTARRAY}(e_1, e_2, e_3) \Downarrow Av(\lfloor v_3 \rfloor, \lfloor v_2 \rfloor, [[v_1 \mid i \leq v_2, j \leq v_3]]), 1 + c_1 + c_2 + c_3 + v_3 \cdot v_2} \text{ (const-array)}$$

$$\frac{\sigma, \alpha \vdash e_0 \Downarrow s, c_0 \qquad s \in Number \wedge 1 \leq s < n + 1 \qquad \sigma, \alpha \vdash e_{\lfloor s \rfloor} \Downarrow v_s, c_s}{\sigma, \alpha \vdash \texttt{CHOOSE}(e_0, e_1, \ldots, e_n) \Downarrow v_s, 1 + c_0 + c_s} \text{ (choose)}$$

$$\frac{\sigma, \alpha \vdash e \Downarrow Av(w, h, [[v_{ij}]]), c}{\sigma, \alpha \vdash \texttt{COLUMNS}(e) \Downarrow w, 1 + c} \text{ (columns)}$$

$$\frac{\begin{array}{ccc}\sigma, \alpha \vdash e_1 \Downarrow Av(w, h, [[v_{ij}]]), c_1 & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 & \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 \\ & v_2 \in Number \wedge 1 \leq v_2 < w + 1 & v_3 \in Number \wedge 1 \leq v_3 < h + 1\end{array}}{\sigma, \alpha \vdash \texttt{INDEX}(e_1, e_2, e_3) \Downarrow v_{\lfloor v_3 \rfloor \lfloor v_2 \rfloor}, 1 + c_1 + c_2 + c_3} \text{ (index)}$$

$$\frac{\begin{array}{c}\sigma,\alpha\vdash e_1\Downarrow Av(w,h,[[v_{ij}]]),c_1 \quad arr=[[v_{ij}]] \\ \sigma,\alpha\vdash e_2\Downarrow v_2,c_2 \quad v_2\in Number \quad 1\le v_2<h+1 \quad h'=\lfloor v_4\rfloor-\lfloor v_2\rfloor+1 \\ \sigma,\alpha\vdash e_4\Downarrow v_4,c_4 \quad v_4\in Number \quad 1\le v_4<h+1 \\ \sigma,\alpha\vdash e_3\Downarrow v_3,c_3 \quad v_3\in Number \quad 1\le v_3<w+1 \quad w'=\lfloor v_5\rfloor-\lfloor v_3\rfloor+1 \\ \sigma,\alpha\vdash e_5\Downarrow v_5,c_5 \quad v_5\in Number \quad 1\le v_5<w+1 \\ r=Av(w',h',[[arr[i,j]\mid v_3\le i\le v_5,v_2\le j\le v_4]]) \quad c_6=w'\cdot h'\end{array}}{\sigma,\alpha\vdash \texttt{SLICE}(e_1,e_2,e_3,e_4,e_5)\Downarrow r,1+c_1+c_2+c_3+c_4+c_5+c_6}\ \text{(slice)}$$

$$\frac{\sigma,\alpha\vdash e\Downarrow v,c \quad v\in Error}{\sigma,\alpha\vdash \texttt{ISERROR}(e)\Downarrow 1,1+c}\ \text{(iserror-true)}$$

$$\frac{\sigma,\alpha\vdash e\Downarrow v,c \quad v\notin Error}{\sigma,\alpha\vdash \texttt{ISERROR}(e)\Downarrow 0,1+c}\ \text{(iserror-false)}$$

$$\frac{\sigma,\alpha\vdash e\Downarrow v,c \quad v\in ArrVal}{\sigma,\alpha\vdash \texttt{ISARRAY}(e)\Downarrow 1,1+c}\ \text{(isarray-true)}$$

$$\frac{\sigma,\alpha\vdash e\Downarrow v,c \quad v\notin ArrVal}{\sigma,\alpha\vdash \texttt{ISARRAY}(e)\Downarrow 0,1+c}\ \text{(isarray-false)}$$

$$\frac{\sigma,\alpha\vdash e_1\Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha\vdash e_n\Downarrow v_n,c_n}{\sigma,\alpha\vdash \texttt{MAX}(e_1,\ldots,e_n)\Downarrow max(v_1,\ldots,v_n),1+\sum_{j=1}^{n}c_j}\ \text{(max)}$$

$$\frac{\sigma,\alpha\vdash e\Downarrow Av(w,h,[[v_{ij}]]),c \quad arr=[[v_{ij}]]}{\sigma,\alpha\vdash \texttt{TRANSPOSE}(e)\Downarrow Av(h,w,[[arr[j,i]\mid i\le h,j\le w]]),1+c+w\cdot h}\ \text{(transpose)}$$

$$\frac{\sigma,\alpha\vdash e_1\Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha\vdash e_n\Downarrow v_n,c_n}{\sigma,\alpha\vdash \texttt{AVERAGE}(e_1,\ldots,e_n)\Downarrow \frac{1}{n}\sum_{i=1}^{n}v_i,1+\sum_{i=1}^{n}c_i}\ \text{(average)}$$

$$\frac{\sigma,\alpha\vdash e_1\Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha\vdash e_n\Downarrow v_n,c_n}{\sigma,\alpha\vdash \texttt{HARRAY}(e_1,\ldots,e_n)\Downarrow Av(n,1,[[v_1,\ldots,v_n]]),1+\sum_{i=1}^{n}c_i+n}\ \text{(harray)}$$

$$\frac{\sigma,\alpha\vdash e_1\Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha\vdash e_n\Downarrow v_n,c_n \quad w=\sum_{i=1}^{n}width(v_i) \quad \forall i,j.\,height(v_i)=height(v_j)}{\sigma,\alpha\vdash \texttt{HCAT}(e_1,\ldots,e_n)\Downarrow Av(w,height(v_1),[[v_1:v_2:\ldots:v_n]]),1+\sum_{i=1}^{n}c_i+n}\ \text{(hcat)}$$

**Figure 20:** Operational and cost semantics for a subset of Funcalc's built-in, first-order functions. For rules that are trivially similar such as those for `ASIN`, `ACOS` and `ATAN`, we omit repetitions and give just a single rule to represent them all.

Rule (now) has one premise stating that if the result may evaluate to a number, the call may evaluate to value $v$ at cost 1 where $v$ is the number of fractional days since the $30^{\text{th}}$ of December, 1899.

Rule (pi) states that it may evaluate to a value $v$ at cost 1, given that $v$ is a number and is equal to $\pi$.

Rule (na) states that the function application of `NA` may evaluate to the error #NA at cost 1.

Rule (abs) states that if $e$ may evaluate to the number $v$ at cost $c$ then the call may evaluate to the absolute value of $v$.

Rule (asin) is similar to rule (abs) but may instead evaluate to the result of a call to the actual inverse trigonometric function *asin*.

Rules (not-1) and (not-2) handle the two different outcomes of the `NOT` function (barring error values in the arguments). Rule (not-1) states that if $e$ may evaluate to zero at some cost then the call may evaluate to one. Rule (not-2) handles the case where the value is different from zero in which case the result may evaluate to zero.

Rule (ceiling) states that if its two argument expressions may evaluate to numbers then the conclusion may evaluate to a call to *ceiling* with the two numbers as arguments.

Rule (equal) is akin to rule (ceiling) except that it may evaluate to the equality comparison between the two numbers. The actual implementation of equality is slightly more involved also taking `null` values and object equality into consideration. We leave out the rules for other comparisons here.

Rules (and-false) and (and-true) borrow notation from Section 4.1. One may pick some subset of indices $J$ where all the corresponding expressions may evaluate to numbers. If there exists an index for which the expression may evaluate to zero, the result of calling `AND` is zero. Rule (and-true) handles the case where all the evaluated expressions may evaluate to a non-zero number value. The rule for `OR` is analogous but swaps true (non-zero) and false (zero) everywhere. These rules admit both sequential strict evaluation, sequential non-strict evaluation, and parallel evaluation of the subexpressions. The total work is proportional to the subset of expressions evaluated plus one.

Rule (sum) says that if all its argument expressions evaluate to number values then the function call may evaluate to the sum of those values. In Funcalc, functions like `SUM` and `AVERAGE` can accept a combination of numbers and array values. The result of the calls to `SUM` are all 21 in the following examples. We choose not to complicate the rules further but one could imagine some sort

of flattening function *flatten* applied to each value in the summation in the conclusion of rule (sum) to account for array values.

```
=SUM(1, 2, 3, 4, 5, 6)
=SUM(HCAT(1, 2, 3, 4, 5, 6))
=SUM(VCAT(1, 2, 3, 4, 5, 6))
=SUM(HCAT(1, 2), 3, VCAT(4, 5, 6))
```

Rule (const-array) says that if expression $e_1$ may evaluate to a value at some cost $c_1$ and expressions $e_2$ and $e_3$ may evaluate to non-negative numbers, then the call may evaluate to an array value of size $v_3 \cdot v_2$ with $v_1$ as the values of each element. The cost reflects that it is only necessary to evaluate $e_1$ once.

Rule (choose) states that if $e_0$ may evaluate to a positive number $s \in [1, n+1[$ and the $i^{th}$ expression, where $i = \lfloor s \rfloor$, may evaluate to a value $v_i$ at cost $c_i$, then the call may evaluate to $v_i$ at cost $1 + c_0 + c_i$. Note that $s$ may lie in a wider interval than required since the current implementation truncates floating-point numbers to integers, hence the floor notation on the subscript $e_{\lfloor s \rfloor}$ etc. Since `CHOOSE` is non-strict, we require only that evaluation of $e_i$ takes place. A more general rule would adopt the same approach we used for the rules for `AND` and have $J = \{0, i\}$ as a special case.

Rule (columns) states that if $e$ may evaluate to an array value then a call to `COLUMNS` may evaluate to the width of that array value. Notice that the work is overly pessimistic. Any sensible implementation would strive to perform a single lookup operation on the array without having to evaluate the entire array first.

Rule (index) states that if $e_1$ may evaluate to an array value and $e_2$ and $e_3$ may evaluate to numbers within the bounds of the array value, then the conclusion may evaluate to the value at index $(\lfloor v_3 \rfloor, \lfloor v_2 \rfloor)$. Like rule (columns), the work is overly pessimistic and like rule (choose) the indices are truncated towards zero.

In rule (slice), the premises state that $e_1$ may evaluate to an array value, expressions $e_2$ and $e_4$ may evaluate to a start- and end column index and expressions $e_3$ and $e_5$ may evaluate to a start- and end row index, where the indices delimit a sub-array within the input array. The conclusion may then evaluate to a new array value that is a slice of the original array. The sub-array's size is computed from the row and column indices. The work is one plus evaluating the four indices plus the work of evaluating the input array value plus the size of the new array.

Rule (iserror-true) states that if $e$ may evaluate to a value $v \in \mathit{Error}$ then the call may evaluate to 1. Rule (iserror-false) is complementary and handles the case where $v \notin \mathit{Error}$.

Rule (isarray-true) and (isarray-false) are analogous to rules (iserror-true) and (iserror-false) but check whether the argument is an array value.

Rule (max) states that if all the argument expressions may evaluate to numbers at some corresponding costs, then the call may evaluate to the maximal value of those values. The rule for `MIN` is analogous.

Rule (transpose) states that if the argument expression may evaluate to an array value of size $w \cdot h$ with cost $c$, then the call may evaluate to a transposed array value of size $h \cdot w$. Notice that element access has been swapped to $v_{ji}$. The work is one plus the cost $c$ and the size of the resultant array.

Rule (average) is similar to rule (sum) but the conclusion may instead evaluate to the average of the input values or an error if there are no input values.

Rule (harray) states that if the arguments may evaluate to a set of values and associated costs, then the call may evaluate to a single-row array of those values. This is consistent with the behaviour of `HARRAY` which puts the values of the evaluated expressions inside an array. The expression `=HARRAY(1, HARRAY(2, 3))` will yield an array value of width 2 and height 1 where the first element is the value 1 and the second element is an array of the same size with values 2 and 3. The rule for `VARRAY` is similar and has been omitted. The $n$ in the cost of the conclusion denotes the cost of allocating the new array.

Rule (hcat) is closely related to the rule for `HARRAY` but concatenates its arguments, which is why there are additional premises to ensure correct dimensions of the argument expressions to `HCAT`. Its premises state that the expressions may evaluate to values at some associated costs as in rule (harray). The width of the new array value is the sum of the widths of all its arguments. The function *width* is defined as follows.

$$width(v) = \begin{cases} w & \text{if } v = Av(w, h, [[v_{ij}]]) \\ 1 & \text{otherwise} \end{cases}$$

We also require that any pair of evaluated expressions must have the same height, otherwise we would not be able to properly concatenate them. The definition of *height* is analogous to that of *width*. Given these premises, the conclusion may evaluate to an array value of width $w$ and the height of the arguments. The elements of the array value are the concatenation of the evaluated expressions. The $n$ in the cost of the conclusion denotes the cost of concatenation and assumes an efficient implementation of array concatenation. To understand the difference between `HCAT` and `HARRAY`, calling the same expression as before with `HCAT`, i.e. `=HCAT(1, HARRAY(2, 3))`, yields an array value of width 3 and height 1. This would also be the case if we replaced the inner call to `HARRAY` with a call to `HCAT`. The rule for `VCAT` is similar and has been omitted for brevity.

## 6.2 Rules for Higher-Order Intrinsic Functions

The notation for higher-order functions builds on rule (g5v) from Section 5. One significant difference is that since most higher-order functions call the supplied function multiple times, we introduce quantification over the environment $\rho'$. For example, the function `TABULATE` calls a function for each position in the result array where we quantify over the fresh environment with the current position $(i, j)$ as $\rho'_{ij}$. Recall that $\rho'$ can be thought of as a stack frame for a function application. Similarly, we also quantify over the cost environment $\gamma'$ as $\gamma'_{ij}$. We start by introducing the rule for `TABULATE` in full detail, then define appropriate auxiliary functions so that we do not need to repeat ourselves in the remaining rules.

$$\frac{\begin{array}{c}\sigma, \alpha \vdash e_1 \Downarrow FunVal(sdf, [u_1, \ldots, u_k]), c_1 \\ def(sdf) = (out, [in_1, \ldots, in_{k+2}], cells) \\ \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 \quad v_2 \in Number \wedge h = \lfloor v_2 \rfloor \geq 0 \\ \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 \quad v_3 \in Number \wedge w = \lfloor v_3 \rfloor \geq 0 \\ \forall i, j.\, \rho'_{ij}(in_1) = u_1 \quad \ldots \quad \rho'_{ij}(in_k) = u_k \quad \rho'_{ij}(in_{k+1}) = i \quad \rho'_{ij}(in_{k+2}) = j \\ \forall i, j.\, \forall ca \in dom(\rho'_{ij}) \setminus \{in_1, \ldots, in_{k+2}\}.\, \sigma, \alpha \vdash \phi(ca) \Downarrow \rho'_{ij}(ca), \gamma'_{ij}(ca) \\ \forall i, j.\, v_{ij} = \rho'_{ij}(out) \quad c_4 = \sum_{i,j}^{dom(\gamma'_{ij})} \sum_{ca} \gamma'_{ij}(ca)\end{array}}{\sigma, \alpha \vdash \texttt{TABULATE}(e_1, e_2, e_3) \Downarrow Av(v_3, v_2, [[v_{ij}]]), 1 + c_1 + c_2 + c_3 + c_4} \text{ (tabulate)}$$

Considering the premises from top to bottom, they state that $e_1$ may evaluate to a function value (at cost $c_1$) that expects two more arguments, and that $e_2$ and $e_3$ may evaluate to non-negative numbers $h$ of rows and $w$ of columns, after truncation towards zero. We then postulate $w \cdot h$ environments $\rho'_{ij}$ where $i \leq w$ and $j \leq h$, one for each application of the function value. The first quantified premise states that the input cells should contain the bound values $[u_1, \ldots, u_k]$ except that the last two arguments must be the indices $i$ and $j$ of the function application. The second quantified premise states that for all cell addresses $ca$ in the domain of $\rho'_{ij}$, excluding the set of input cells, the expression of that cell address may evaluate to the value given by environment $\rho'_{ij}$ at some cost. The final premise states that each function application evaluates to the value $v_{ij}$ of the function call's output cell. It is intuitive to think of each $\rho'_{ij}$ as a stack frame for a function application at a position $(i, j)$. The call to `TABULATE` then evaluates to an array value of size $w \cdot h$ whose elements are $v_{ij}$. The work is 1 plus the work for evaluating the function value, the dimension expressions, and the sum of the costs of applying the *sdf* for every index combination $(i, j)$.

The function application notation in the right-hand side column can be abstracted away since this is how all higher-order functions apply function values, so in an effort to reduce repetition, we define a function *apply* to be definitionally equal to the following definition.

$$apply_{\sigma,\alpha}(sdf, [u_1, \ldots, u_k], a_0, \ldots, a_n, r, c) \triangleq \begin{cases} \rho'(in_1) = u_1 \ \ldots \ \rho'(in_k) = u_k \quad \rho'(in_{k+1}) = a_0 \ \ldots \ \rho'(in_{k+n}) = a_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}.\, \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca) \\ r = \rho'(out) \\ c = \sum_{ca}^{dom(\gamma')} \gamma'(ca) \end{cases}$$

Note that the resulting value $r$ and cost $c$ are passed back out of the apply function definition. This definition makes sense for functions like `TABULATE` where we can refer to specific function applications using $apply_{\sigma,\alpha}$ in terms of its position in the resultant array by using $r_{ij}$ and $c_{ij}$. However, for recursive functions like `REDUCE`, we introduce a new judgement form $\sigma, \alpha \vdash_v s \Downarrow v, c$ that operates on values instead of expressions in order to handle the intermediate computations of `REDUCE` that operate on values. The judgement states that given the usual environments $\sigma$ and $\alpha$, some intermediate value-based computation state $s$ may evaluate to a value $v$ at cost $c$. This allows us to evaluate the argument expressions $e_1, e_2$ and $e_3$ at the top-level call once and then use their values in subsequent recursive calls.

$$\frac{\begin{array}{lll} \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 & v_1 = FunVal(sdf, [u_1, \ldots, u_k]) & \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3 \\ & k = arity(sdf) - 2 & v_3 \in ArrVal \\ & \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 & \sigma, \alpha \vdash_v \texttt{REDUCE}(v_1, v_2, v_3) \Downarrow r, c_r \end{array}}{\sigma, \alpha \vdash \texttt{REDUCE}(e_1, e_2, e_3) \Downarrow r, 1 + c_1 + c_2 + c_3 + c_r} \text{ (reduce)}$$

We start with the expression-based rule (reduce) for the `REDUCE` function. Expressions $e_1$ and $e_2$ may evaluate to a function value and an initial value for the reduction, respectively. Expression $e_3$ may evaluate to an array $v_3$ which is passed as an argument to the value-based reduction rule. The conclusion may then evaluate to the result $r$ of the value-based reduction. Next, we define the inductive and base rules for the value-based reduction.

$$\frac{\begin{array}{ll}\sigma, \alpha \vdash_v v_3 = v_l : v_r, c_d & v_l \in ArrVal \wedge v_r \in ArrVal \\ v_1 \in FunVal & \sigma, \alpha \vdash_v \texttt{REDUCE}(v_1, v_2, v_l) \Downarrow r_l, c_l \\ v_2 \in Number & \sigma, \alpha \vdash_v \texttt{REDUCE}(v_1, r_l, v_r) \Downarrow r, c_r\end{array}}{\sigma, \alpha \vdash_v \texttt{REDUCE}(v_1, v_2, v_3) \Downarrow r, 1 + c_3 + c_l + c_r + c_d} \text{ (reduce-inductive)}$$

The recursive, value-based reduction rule (reduce-inductive) first decomposes $v_3$ into two arrays $v_l$ and $v_r$ at some cost $c_d$. This can be an arbitrary decomposition as chosen by an implementation since, given an identity element and an associative, binary function, a reduction may e.g. proceed from left to right using a decomposition similar to functional lists; or by decomposing the operations as a tree by recursively splitting the input array in halves, we can perform the reduction in parallel. Notice that we pass the result of the reduction of the left decomposed array $v_l$ to the reduction of the right decomposed array $v_r$. The reason is purely semantic and will become apparent shortly.

We need two additional base case rules to account for a reduction of an odd number of values and for the empty list. These are given as rules (reduce-base-odd) and (reduce-base-empty).

$$\frac{\begin{array}{ll}v_1 \in FunVal & v_3 = Av(1, 1, [[v_{11}]]) \\ apply_{\sigma,\alpha}(sdf, [u_1, \ldots, u_k], v_2, v_{11}, r, c)\end{array}}{\sigma, \alpha \vdash_v \texttt{REDUCE}(v_1, v_2, v_3) \Downarrow r, 1 + c} \text{ (reduce-base-odd)}$$

$$\frac{v_1 \in FunVal \quad v_2 \in Number \quad v_3 = Av(0, 0, [[]])}{\sigma, \alpha \vdash_v \texttt{REDUCE}(v_1, v_2, v_3) \Downarrow v_2, 1} \text{ (reduce-base-empty)}$$

Rule (reduce-base-odd) handles the case where a single-element array is given by $v_3$ where we apply the sheet-defined function *sdf* from the function value $v_1$ to the starting value $v_2$ and the single element $v_{11}$ of $v_3$. In rule (reduce-inductive), if we did not thread the result through the left and right decomposition of the array argument as mentioned earlier, rule (reduce-base-odd) might be applied more than once which in turn would cause the starting value $v_2$ to also be used more than once, resulting in an incorrect result. Rule (reduce-base-empty) returns the starting value $v_2$ of the reduction if passed the empty array.

To illustrate these rules, we expand the derivation tree for the following expression in Figure 21 and show the corresponding tree decomposition in Figure 22.

In the example, $c_d$ denotes the cost of array decomposition, $c_a$ the cost of applying the addition operator, and $c_h$ denotes the cost of concatenating elements with `HCAT`. Additionally, we have omitted some of the set membership tests to keep the derivation tree succinct.

```
=REDUCE(CLOSURE("+"), 8, HCAT(1, 2, 3, 4))
```

The result of the expression is $8 + 1 + 2 + 3 + 4 = 18$. For brevity, we denote lists in square braces $[1, 2, 3, 4]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{apply_{\sigma,\alpha}(sdf, [] , 8, 1, r, c_a)}}{\sigma, \alpha \vdash_v \texttt{REDUCE}(Fv(sdf, []), 8, [1]) \Downarrow 9, 1 + c_a} \quad \dfrac{}{apply_{\sigma,\alpha}(sdf, [] , 9, 2, r, c_a)}}{\sigma, \alpha \vdash_v \texttt{REDUCE}(Fv(sdf, []), 9, [2]) \Downarrow 11, 1 + c_a}}{\sigma, \alpha \vdash_v [1, 2] \Downarrow [1] : [2] , c_d}}{\sigma, \alpha \vdash_v \texttt{REDUCE}(Fv(sdf, []), 8, [1, 2]) \Downarrow 11, 1 + 2 + c_d + 2c_a}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{apply_{\sigma,\alpha}(sdf, [] , 11, 3, r, c_a)}}{\sigma, \alpha \vdash_v \texttt{REDUCE}(Fv(sdf, []), 11, [3]) \Downarrow 14, 1 + c_a} \quad \dfrac{}{apply_{\sigma,\alpha}(sdf, [] , 14, 4, r, c_a)}}{\sigma, \alpha \vdash_v \texttt{REDUCE}(Fv(sdf, []), 14, [4]) \Downarrow 18, 1 + c_a}}{\sigma, \alpha \vdash_v [3, 4] \Downarrow [3] : [4] , c_d}}{\sigma, \alpha \vdash_v \texttt{REDUCE}(Fv(sdf, []), 11, [3, 4]) \Downarrow 18, 1 + 2 + c_d + 2c_a}$$

$$\dfrac{\dfrac{\sigma, \alpha \vdash_v [1, 2, 3, 4] \Downarrow [1, 2] : [3, 4] , c_d}{\sigma, \alpha \vdash_v \texttt{REDUCE}(Fv(sdf, []), 8, [1, 2, 3, 4]) \Downarrow 18, 1 + 3 + 3 + 3c_d + 4c_a} \quad \sigma, \alpha \vdash \texttt{CLOSURE("+")} \Downarrow Fv(sdf, []), c_1 \quad \sigma, \alpha \vdash 8 \Downarrow 8, 1 \quad \sigma, \alpha \vdash \texttt{HCAT(1, 2, 3, 4)} \Downarrow [1, 2, 3, 4] , c_h}{\sigma, \alpha \vdash \texttt{REDUCE(CLOSURE("+"), 8, HCAT(1, 2, 3, 4))} \Downarrow 18, 1 + 7 + 3c_d + 4c_a + c_h + c_1}$$

**Figure 21:** The full derivation tree of the expression `=REDUCE(CLOSURE("+"), 8, HCAT(1, 2, 3, 4))` using a combination of the expression- and value-based rules for reduction.

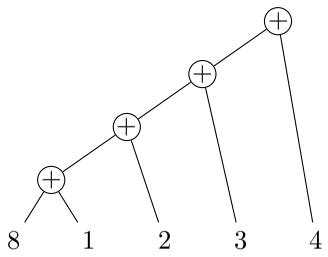


**Figure 22:** The tree for the reduction in the `REDUCE` example.

We are now ready to list the rules for the remaining higher-order functions in Funcalc which are shown in Figure 23.

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_0 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_0 \qquad \sigma,\alpha \vdash e_1 \Downarrow Av(w_1,h_1,[[v_{ij}^1]]),c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow Av(w_n,h_n,[[v_{ij}^n]]),c_n \\ k = arity(sdf)-n \qquad \forall k,m.\, w_k = w_m \wedge h_k = h_m \\ \forall i,j.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],v_{ij}^1,\ldots,v_{ij}^n,r_{ij},t_{ij})\end{array}}{\sigma,\alpha \vdash \texttt{MAP}(e_0,e_1,\ldots,e_n) \Downarrow Av(w_1,h_1,[[r_{ij}]]),1+c_0+\sum_{k=1}^n c_k+\sum_{ij} t_{ij}} \text{(map)}$$

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_1 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_1 \qquad \sigma,\alpha \vdash e_2 \Downarrow Av(w,h,[[v_{ij}]]),c_2 \\ k = arity(sdf)-h \qquad \forall i.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],[[v_{i*}]],r_{i1},c_i)\end{array}}{\sigma,\alpha \vdash \texttt{COLMAP}(e_1,e_2) \Downarrow Av(w,1,[[r_{i1}]]),1+c_1+c_2+\sum_i c_i} \text{(colmap)}$$

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_0 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_0 \qquad \sigma,\alpha \vdash e_1 \Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow v_n,c_n \\ k = arity(sdf)-1 \qquad \forall i.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],v_i,r_i,t_i)\end{array}}{\sigma,\alpha \vdash \texttt{COUNTIF}(e_0,e_1,\ldots,e_n) \Downarrow \sum\{1 \mid r_i = 1 \wedge i = \{1,\ldots,n\}\},1+c_0+\sum_{j=1}^n c_j+\sum_{i=1}^n t_i} \text{(countif)}$$

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_0 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_0 \qquad \sigma,\alpha \vdash e_1 \Downarrow v_1,c_1 \quad \ldots \quad \sigma,\alpha \vdash e_n \Downarrow v_n,c_n \\ k = arity(sdf)-1 \qquad \forall i.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],v_i,r_i,t_i)\end{array}}{\sigma,\alpha \vdash \texttt{SUMIF}(e_0,e_1,\ldots,e_n) \Downarrow \sum\{v_i \mid r_i = 1 \wedge i = \{1,\ldots,n\}\},1+c_0+\sum_{j=1}^n c_j+\sum_{i=1}^n t_i} \text{(sumif)}$$

$$\frac{\begin{array}{c}\sigma,\alpha \vdash e_1 \Downarrow FunVal(sdf,[u_1,\ldots,u_k]),c_1 \qquad \sigma,\alpha \vdash e_3 \Downarrow v_3,c_3 \\ k = arity(sdf)-1 \qquad v_3 \in Number \\ \sigma,\alpha \vdash e_2 \Downarrow Av(1,h,[[v_{ih}]]),c_2 \qquad w' = \lfloor v_3 \rfloor + 1 \\ r_0 = [[v_{0*}]] \qquad \forall i = 1,\ldots,w'.\, apply_{\sigma,\alpha}(sdf,[u_1,\ldots,u_k],r_{i-1},r_i,t_i) \\ \sigma,\alpha \vdash_v Av(w',h,[[r_0 : \ \ldots \ : r_n]]) \Downarrow arr,c_4\end{array}}{\sigma,\alpha \vdash \texttt{HSCAN}(e_1,e_2,e_3) \Downarrow arr,1+c_1+c_2+c_3+c_4+\sum_{i=1}^n t_i} \text{(hscan)}$$

**Figure 23:** Operational and cost semantics for a subset of Funcalc's higher-order, built-in functions. For rules that are trivially similar such as those for `HCAT`, and `VCAT`, we omit repetitions and give just a single rule to represent them all.

Rule (map) shows the rule for the intrinsic `MAP` function which is in fact a

generalised n-ary `zip` function. Given one argument, `MAP` behaves as a regular mapping function. The rule states that $e_0$ may evaluate to a function value with $k$ bound arguments. Each of the other arguments $e_1, \ldots, e_n$ may evaluate to array values of equal size. The function is applied to each element in all the array value arguments at a given position $(i, j)$ and the result $r_{ij}$ at cost $t_{ij}$ is the value of the resulting array value in the conclusion of the rule at the same position. The total cost is one plus the cost of evaluating the function value, the cost of evaluating all the array value arguments and the cost of all $i \cdot j$ function applications.

Rule (colmap) states that if the first argument may evaluate to a function value with an arity equal to the height of the array value of $e_2$ and that the second may evaluate to an array value, then the call may evaluate to a new single-row, array value where each element is the function application of *sdf* to each column in the input array.

Rule (countif) states that if the first argument $e_0$ may evaluate to an unary function value and the remaining expressions to some values, then a call to `COUNTIF` may evaluate to a sum of ones for which $r_i = 1$ (true).

Rule (sumif) closely resembles rule (countif), but the resulting value in the conclusion may instead evaluate to a summation of the values $v_i$ where $r_i = 1$ (true).

Finally, we have the rule for `HSCAN`. The function performs a column-wise scan operation as opposed to an element-wise scan as per Blelloch [2]. The rule is rather complicated, so an example is in order. Given a function $f(a) = \text{map}(\text{fun } x \Rightarrow x + 1, a)$ and calling the function as

```
HSCAN(CLOSURE("f"), A1:A2, 2)
```

in Figure 24, produces the values in cell area A4:C5. The row values in each column are one greater than the row values of the preceding column. The premises in rule (hscan) say that $e_1$ may evaluate to a function value accepting one argument, $e_2$ may evaluate to a column array value, $e_3$ may evaluate to a number. We introduce special syntax for array values so we can refer to entire rows or columns. Thus, $[[v_{1*}]]$ refers to the first column of an array value whereas $[[v_{*3}]]$ refers to the third row. The result array $arr$ may then evaluate to an array value where each column is the function applied 0 to $n$ times to the input column and then concatenated using the colon operator. Since a function applied zero times is the identity function, the first column is just the original input column which is why the number of columns in $arr$ is $v_3 + 1$.

In any sensible implementation of `HSCAN`, we would avoid the quadratic work of multiple redundant function applications and use the results of previous columns. For example, applying the function once at column $i$ to the result of the previous column $i - 1$ corresponds to having applied the function $i$ times

to the original input column. This is reflected in the quantified premise in the rule.

| | A | B | C |
|---|---|---|---|
| 1 | 1 | | |
| 2 | 2 | | |
| 3 | | | |
| 4 | 1 | 2 | 3 |
| 5 | 2 | 3 | 4 |

**Figure 24:** Column-wise scan using `HSCAN`. Each value from the preceding column is incremented by one.

# 7 Concrete cost calculation functions

In this section we follow Gomez et al. [4], inspired by Rosendahl [5], and present a set of functions which can be used to calculate the actual cost of executing an evaluation of a spreadsheet expression, including higher order functions definable in Funcalc. The idea in the first set of functions is that each function will both evaluate the expression and simultaneously record the computation costs.

$$
\begin{aligned}
T_v[n] &= n \\
T_v[ca] &= ca \\
T_v[IF(e_1, e_2, e_3))] &= IF(T_v[e_1], T_v[e_2], T_v[e_3]) \\
T_v[RAND()] &= RAND() \\
T_v[F(e_1, \ldots, e_n)] &= F(T_v[e_1], \ldots, T_v[e_n]) \\
T_v[ca_1 : ca_2] &= ca_1 : ca_2 \\
T_v[e[i, j]] &= T_v[e][i, j] \\
T_v[sdf(e_1, \ldots, e_n)] &= sdf(T_v[e_1], \ldots, T_v[e_n]) \\
T_v[\texttt{CLOSURE}(sdf, e_1, \ldots, e_n)] &= pair(\texttt{CLOSURE}(sdf, T_v[e_1], \ldots, T_v[e_n]), \\
&\qquad \texttt{CLOSURE}(sdf_t(T_v[e_1], \ldots, T_v[e_n]))) \\
T_v[\texttt{APPLY}(e_0, e_1, \ldots, e_n)] &= fst(T_v[e_0])(T_v[e_1], \ldots, T_v[e_n])
\end{aligned}
$$

$$T_v[def(sdf) = (out, [in_1, \ldots, in_n], cells)] = def(sdf) = (T_v[out], [in_1, \ldots, in_n], T_v[cells])$$

$$
\begin{aligned}
T_t[n] &= 1 \\
T_t[ca] &= 1 \\
T_t[IF(e_1, e_2, e_3)) &= IF(e_1, 1 + T_t[e_1] + T_t[e_2], 1 + T_t[e_1] + T_t[e_3]) \\
T_t[RAND()] &= 1 \\
T_t[F(e_1, \ldots, e_n)] &= F_t(T_v[e_1], \ldots, T_v[e_n]) + 1 + \sum_{i=1,n} T_t[e_i] \\
T_t[ca_1 : ca_2] &= 1 \\
T_t[e[i, j]] &= 1 \\
T_t[sdf(e_1, \ldots, e_n)] &= 1 + \sum_{i=1,n} T_t[e_i] + sdf_t(T_v[e_1], \ldots, T_v[e_n]) \\
T_t[\texttt{CLOSURE}(sdf, e_1, \ldots, e_n)] &= 1 + \sum_{i=1,n} T_t[e_i] \\
T_t[\texttt{APPLY}(e_0, e_1, \ldots, e_n)] &= 1 + T_t[e_0] + \sum_{i=1,n} T_t[e_i] + \texttt{APPLY}(snd(T_v[e_0]), T_v[e_1], \ldots, T_v[e_n])
\end{aligned}
$$

$$T_t[def(sdf) = (out, [in_1, ..., in_n], cells)] = def(sdf_t) = (T_t[out], [in_1, ..., in_n], T_t[cells])$$

Note that the above definition introduces a pair data structure. This data structure could be encoded via cell arrays with two elements. $pair$, $fst$, $snd$ are handled by the semantic rules for built-in functions and they will thus have $pair_t$, $fst_t$ and $snd_t$ functions returning their cost during evaluation.

Conjecture:

$$\sigma, \alpha \vdash e \Downarrow_t v, t \texttt{ iff } \sigma, \alpha \vdash T_v[e] \Downarrow_s v \wedge \sigma, \alpha \vdash T_t[e] \Downarrow_s t$$

Proof:

Straightforward exercise based on structural induction.

This shows that the cost semantics, as defined in Section 4 and denoted here by ($\Downarrow_t$), is preserved by the translation and yields the same values, when the standard semantics, as defined in Section 3.1 and denoted here by ($\Downarrow_s$), is used on the translated terms (i.e. we have used spreadsheet computations to calculate the cost of spreadsheet computations).

It is possible to avoid introducing $pair$, $fst$, $snd$ and thereby simplifying the cost evaluation function compared to the one above which is inspired by [4]. Since Funcalc does not have anonymous functions (i.e. lambdas), but only has named functions, we know which function we need to invoke when we apply a `CLOSURE`

since the name of the function (sdf) is recorded in the `CLOSURE`: `CLOSURE`(sdf, $e_1, \ldots, e_n$).

Since sheet-defined functions are defined as $def(sdf) = (out, [in_1, \ldots, in_n], cells)$, their cost functions can easily be found via a simple naming scheme and translation of the body of the sheet-defined function defined as: $T_t[def(sdf) = (out, [in_1, \ldots, in_n], cells)] = def(sdf_t) = (out, [in_1, \ldots, in_n], T_t[cells])$. All we need to do in the cost translation is to replace $sdf$ with $sdf_t$ in the `CLOSURE` in $e_0$ in the rule for $T_t[\texttt{APPLY}(e_0, e_1, \ldots, e_n)]$.

The following definitions capture this. First we define a function $T_d$ going structurally through a Funcalc expression, replacing all $sdf$ names with their time counterpart $sdf_t$:

$$
\begin{aligned}
T_d[n] &= n \\
T_d[ca] &= ca \\
T_d[IF(e_1, e_2, e_3))] &= IF(T_d[e_1], T_d[e_2], T_d[e_3]) \\
T_d[RAND()] &= RAND() \\
T_d[F(e_1, \ldots, e_n)] &= F(T_d[e_1], \ldots, T_d[e_n]) \\
T_d[ca_1 : ca_2] &= ca_1 : ca_2 \\
T_d[e[i, j]] &= T_d[e][i, j] \\
T_d[sdf(e_1, \ldots, e_n)] &= sdf_t(T_d[e_1], \ldots, T_d[e_n]) \\
T_d[\texttt{CLOSURE}(sdf, e_1, \ldots, e_n)] &= \texttt{CLOSURE}(sdf_t, T_d[e_1], \ldots, T_d[e_n]) \\
T_d[\texttt{APPLY}(e_0, e_1, \ldots, e_n)] &= \texttt{APPLY}(T_d[e_0], T_d[e_1], \ldots, T_d[e_n])
\end{aligned}
$$

This function is then used in the `APPLY` clause of the modified version of $T_t$:

$$
\begin{aligned}
T_t[n] &= 1 \\
T_t[ca] &= 1 \\
T_t[IF(e_1, e_2, e_3)] &= IF(e_1, 1 + T_t[e_1] + T_t[e_2], 1 + T_t[e_1] + T_t[e_3]) \\
T_t[RAND()] &= 1 \\
T_t[F(e_1, \ldots, e_n)] &= F_t(T_v[e_1], \ldots, T_v[e_n]) + 1 + \sum_{i=1,n} T_t[e_i] \\
T_t[ca_1 : ca_2] &= 1 \\
T_t[e[i, j]] &= 1 \\
T_t[sdf(e_1, \ldots, e_n)] &= 1 + \sum_{i=1,n} T_t[e_i] + sdf_t(T_v[e_1], \ldots, T_v[e_n]) \\
T_t[\texttt{CLOSURE}(sdf, e_1, \ldots, e_n)] &= 1 + \sum_{i=1,n} T_t[e_i] \\
T_t[\texttt{APPLY}(e_0, e_1, \ldots, e_n)] &= 1 + T_t[e_0] + \sum_{i=1,n} T_t[e_i] + \\
&\quad \texttt{APPLY}(T_d((T_v[e_0])), T_v[e_1], \ldots, T_v[e_n])
\end{aligned}
$$

$$T_t[def(sdf) = (out, [in_1, \ldots, in_n], cells)] = def(sdf_t) = (T_t[out], [in_1, \ldots, in_n], T_t[cells])$$

$T_v$ in the above definition turns out to be the identity function, thus the definition of $T_t$ can be further simplified:

$$
\begin{aligned}
T_t[n] &= 1 \\
T_t[ca] &= 1 \\
T_t[IF(e_1, e_2, e_3)] &= IF(e_1, 1 + T_t[e_1] + T_t[e_2], 1 + T_t[e_1] + T_t[e_3]) \\
T_t[RAND()] &= 1 \\
T_t[F(e_1, \ldots, e_n)] &= F_t(e_1, \ldots, e_n) + 1 + \sum_{i=1,n} T_t[e_i] \\
T_t[ca_1 : ca_2] &= 1 \\
T_t[e[i, j]] &= 1 \\
T_t[sdf(e_1, \ldots, e_n)] &= 1 + \sum_{i=1,n} T_t[e_i] + sdf_t(e_1, \ldots, e_n) \\
T_t[\texttt{CLOSURE}(sdf, e_1, \ldots, e_n)] &= 1 + \sum_{i=1,n} T_t[e_i] \\
T_t[\texttt{APPLY}(e_0, e_1, \ldots, e_n)] &= 1 + T_t[e_0] + \sum_{i=1,n} T_t[e_i] + T_d[e_0](e_1, \ldots, e_n)
\end{aligned}
$$

$$T_t[def(sdf) = (out, [in_1, \ldots, in_n], cells)] = def(sdf_t) = (out, [in_1, \ldots, in_n], T_t[cells])$$

This implies that it is possible to implement the cost calculation function in Funcalc without changing the implementation of the evaluation method already implemented, as $T_v$ and $T_t$ are no longer mutually recursive and only $T_t$ depends on $T_v$.

# 8 Abstract Cost Semantics

Gomez et al. and Rosendahl worked on cost translations for higher-order functional languages [4, 5] which cater for computing with unknown data values. Adding such unknown values allows for a rudimentary abstract interpretation of programs which in many cases can provide a rather precise approximation of the actual cost of the computation.

Note that is there is no way to insert an unknown value in a sheet. If a cell is left empty, the set $\{0.0\}$ is returned. This is consistent with the standard semantics. However, it may be interesting to abstractly evaluate sheets where some cells have unknown values. This can be handled by a built-in function `Unknown` taking no arguments and always returning the unknown value, denoted $\top$.

To allow computations with unknown values, we need an abstract representation of values that models the concrete values used in concrete interpretation [3]. Figure 25 gives a suggested lattice of abstract values for Funcalc.

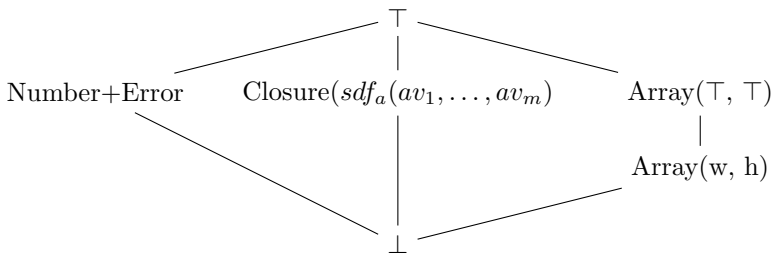


**Figure 25:** A lattice of abstract values that can be used in abstract interpretation of Funcalc. The symbol $\top$ denotes that a value can be represented by multiple values in the lattice, e.g. something is both an atomic value and an array of known size. In type systems, this constitutes a type unification error. The symbol $\bot$ denotes that we know nothing about a value. The values are either the semantic map of numbers and errors, abstract closures, or array values. For arrays, we need both an abstraction for an array of unknown size Array($\top$, $\top$) and an array of known size Array(w, h).

The lattice depicted in Figure 25 can now be used in the definition of abstract values computed by the abstract semantics for Funcalc. These are depicted in Figure 26.

$$
\begin{array}{llll}
n & \in & Number & = \{ \text{ proper numbers } \} \\
absav & \in & AbsArrVal & = \{ (w, h, [[absv_{ij} \mid i \leq w, j \leq h]]) \} + \{(\top, \top)\} \\
absfv & \in & AbsFunVal & = \{ (sdf, [absu_1, \ldots, absu_k]) \} \\
 & & Error & = \{ \texttt{\#DIV/0!}, \texttt{\#CYCLE!} \} \\
ca & \in & Addr & = \{ \text{ cell addresses } \} \\
absv, absu & \in & AbsValue & = Number + Error + AbsArrVal + AbsFunVal + \{\top\} \\
e & \in & Expr & = \{ \text{ formulas, see Figure 5 } \} \\
\phi_a & & & \in Addr \to Expr \\
\sigma_a & & & \in Addr \to AbsValue \\
\alpha_a & & & \in Expr \to AbsValue \\
\rho_a & & & \in Addr \to AbsValue
\end{array}
$$

**Figure 26:** Sets and maps used in the Abstract Funcalc semantics. The tuple $\{(\top, \top)\}$ component of $AbsArrVal$ represents an array of unknown size

The ordering on $AbsValue$ is such that $absv \sqsubseteq \top$ for all $absv \in AbsValue$. Furthermore, $absav \sqsubseteq Array(\top, \top)$ for all $absav \in AbsArrVal$.

We can now follow the ideas presented by Schmidt [3] and provide a trace-based abstract interpretation for Funcalc, based on the ideas for big step semantics presented in section 5 of [3].

The cost semantics for Funcalc presented in Section 5 is extended with the following rules:

$$\frac{\sigma, \alpha \vdash e_1 \Downarrow \top, c_1 \qquad \sigma, \alpha \vdash e_2 \Downarrow v_2, c_2 \qquad \sigma, \alpha \vdash e_3 \Downarrow v_3, c_3}{\sigma, \alpha \vdash \texttt{IF}(e_1, e_2, e_3) \Downarrow v_2 \sqcup v_3, 1 + c_1 + \texttt{max}(c_2, c_3)} \; (g3a)$$

$$\frac{\begin{array}{c} J \subseteq \{1, \ldots, n\} \\ \forall j \in J.\ \sigma, \alpha \vdash e_j \Downarrow v_j, c_j \qquad v_i = \top \text{ for some } i \in J \end{array}}{\sigma, \alpha \vdash \texttt{F}(e_1, \ldots, e_n) \Downarrow \top, 1 + \sum_{j \in J} c_j + work(f, v_1, \ldots, v_n)} \; (g5a)$$

$$\frac{ca \notin dom(\sigma)}{\sigma, \alpha \vdash \texttt{ca} \Downarrow \top, 1} \; (g2a)$$

$$\frac{\begin{array}{c} \sigma, \alpha \vdash e_0 \Downarrow \top, c_0 \\ \sigma, \alpha \vdash e_1 \Downarrow v_1, c_1 \qquad \ldots \qquad \sigma, \alpha \vdash e_n \Downarrow v_n, c_n \\ \forall ca \in dom(\rho') \setminus \{in_1, \ldots, in_{k+n}\}.\ \rho', \sigma \vdash \phi(ca) \Downarrow \rho'(ca), \gamma'(ca) \end{array}}{\sigma, \alpha \vdash \texttt{APPLY}(e_0, e_1, \ldots, e_n) \Downarrow \top, \infty} \; (g10a)$$

With these rules, it is possible to follow [3] and establish a safety property for finite derivations. First we define a safety property for values and costs:

$$v, c \text{ safe}_{val} av, ac \texttt{ iff } v \sqsubseteq av \texttt{ and } c \sqsubseteq ac$$

Here we use the ordering on values defined above and the normal order on costs augmented with $c \leq \infty$ for all $c$.

The safety property on values is then extended to environments:

$$\begin{aligned} \alpha, \rho \text{ safe}_{env} \alpha_a, \rho_a \texttt{ iff } & \texttt{dom}(\alpha) = \texttt{dom}(\alpha_a) \\ \texttt{and } & \texttt{dom}(\rho) = \texttt{dom}(\rho_a) \\ \texttt{and } & \forall ca.\alpha(ca) \text{ safe}_{val} \alpha_a(ca) \\ \texttt{and } & \forall ca.\rho(ca) \text{ safe}_{val} \rho_a(ca) \end{aligned}$$

Finally the safety property can be extended to sequents.

$$\sigma, \alpha \vdash e \Downarrow_t v, c \text{ safe}_{seq} \sigma_a, \alpha_a \vdash e \Downarrow_{at} av, ac \texttt{ iff } \alpha, \rho \text{ safe}_{env} \alpha_a, \rho_a \texttt{ and } v, c \text{ safe}_{val} av, ac$$

With the safety property on sequents we can extend the definition to trees $\text{safe}_{tree}$. $T_C \text{ safe}_{tree} T_A$ holds if $root(T_C) \text{ safe}_{seq} root(T_A)$ and for every child subtree $t_i$ of $T_C$ there exists a subtree $t_j$ of $T_A$ such that $t_i \text{ safe}_{tree} t_j$ holds.

The desired safety property can now be established. For every expression $e$ and concrete environments, respectively abstract environments such that $\alpha, \rho \text{ safe}_{env} \alpha_a, \rho_a$, we can establish that for every proof tree $t_C$ in the concrete semantics with $t_C \in \text{wftree}_C$ and $\text{root}(t_C) = \sigma, \alpha \vdash e \Downarrow_t v, c$ and for every proof tree $t_A$ in the abstract semantics with $t_A \in \text{wftree}_A$ and $\text{root}(t_A) = \sigma_a, \alpha_a \vdash e \Downarrow_{at} av, ac$, it is the case that $t_C \text{ safe}_{tree} t_A$. The proof of this follows by induction on the height of the derivation tree.

The above only establishes a safety property for finite derivations, i.e. for terminating programs. Not all programs terminate and we therefore need to look into handling infinite derivations as well.

# 9 Abstract Cost Calculation Functions

First we need a transformation on expressions capable of calculations with unknown ($\top$):

$$
\begin{aligned}
T_{vb}[n] &= n \\
T_{vb}[ca] &= ca \\
T_{vb}[IF(e_1, e_2, e_3))] &= IF_{vb}(T_{vb}[e_1], T_{vb}[e_2], T_{vb}[e_3]) \\
T_{vb}[RAND()] &= RAND() \\
T_{vb}[F(e_1, \ldots, e_n)] &= F_{vb}(T_{vb}[e_1], \ldots, T_{vb}[e_n]) \\
T_{vb}[ca_1 : ca_2] &= ca_1 : ca_2 \\
T_{vb}[e[i, j]] &= T_{vb}[e][i, j] \\
T_{vb}[sdf(e_1, \ldots, e_n)] &= sdf_t(T_{vb}[e_1], \ldots, T_{vb}[e_n]) \\
T_{vb}[\texttt{CLOSURE}(sdf, e_1, \ldots, e_n)] &= \texttt{CLOSURE}(sdf_{vb}, T_{vb}[e_1], \ldots, T_{vb}[e_n]) \\
T_{vb}[\texttt{APPLY}(e_0, e_1, \ldots, e_n)] &= APPLY_{vb}(T_{vb}[e_0], T_{vb}[e_1], \ldots, T_{vb}[e_n])
\end{aligned}
$$

where

$$
\begin{aligned}
IF_{vb}(e_1, e_2, e_3) &= IF(e_1 = \top, e_2 \sqcup e_3, IF(e_1, e_2, e_3)) \\
APPLY_{vb}(e_0, e_1, \ldots, e_n) &= IF(e_0 = \top, \top, (e_0)(e_1, \ldots, e_n))
\end{aligned}
$$

and

$$T_{vb}[def(sdf) = (out, [in_1, \ldots, in_n], cells)] = def(sdf_{vb}) = (out, [in_1, \ldots, in_n], T_{vb}[cells])$$

Using this we can define an abstract time function $T_{at}$:

$$
\begin{aligned}
T_{at}[n] &= 1 \\
T_{at}[ca] &= 1 \\
T_{at}[IF(e_1, e_2, e_3))] &= IF(T_{vb}(e_1) = \top, max(1 + T_{at}[e_1] + T_t[e_2], 1 + T_{at}[e_1] + T_t[e_3]), \\
&\qquad IF(T_{vb}(e_1), 1 + T_{at}[e_1] + T_t[e_2], 1 + T_{at}[e_1] + T_t[e_3])) \\
T_{at}[RAND()] &= 1 \\
T_{at}[F(e_1, \ldots, e_n)] &= F_{at}(e_1, \ldots, e_n) + 1 + \sum_{i=1,n} T_{at}[e_i] \\
T_{at}[ca_1 : ca_2] &= 1 \\
T_{at}[e[i, j]] &= 1 \\
T_{at}[sdf(e_1, \ldots, e_n)] &= 1 + \sum_{i=1,n} T_{at}[e_i] + sdf_{at}(e_1, \ldots, e_n) \\
T_{at}[\texttt{CLOSURE}(sdf, e_1, \ldots, e_n)] &= 1 + \sum_{i=1,n} T_{at}[e_i] \\
T_{at}[\texttt{APPLY}(e_0, e_1, \ldots, e_n)] &= IF(T_{vb}(e_0) = \top, \infty, 1 + T_{at}[e_0] + \sum_{i=1,n} T_{at}[e_i] + \\
&\qquad \texttt{APPLY}(T_{vb}[e_0], e_1, \ldots, e_n]))
\end{aligned}
$$

$$T_{at}[def(sdf) = (out, [in_1, \ldots, in_n], cells)] = def(sdf_t) = (T_{at}[out], [in_1, \ldots, in_n], T_{at}[cells])$$

We may relate the safety of the abstract semantics described in Section 8 to the abstraction functions defined above:

Conjecture:

$$\sigma_a, \alpha_a \vdash e \Downarrow_{at} av, at \texttt{ iff } \sigma_a, \alpha_a \vdash T_{vb}[e] \Downarrow_s av \wedge \sigma_a, \alpha_a \vdash T_{at}[e] \Downarrow_s at$$

Proof:

Straightforward exercise based on structural induction.

The evaluation of $T_{vb}[e]$ and $T_{at}[e]$ may loop whenever the evaluation of $e$ loops. This may be acceptable as long as the abstract cost is evaluated simultaneously with the (abstract) value of an expression $e$. However, it may be more desirable to further abstract the abstract evaluation to ensure that the abstract cost evaluation always terminates.

```
1 public interface ICellEvaluator<T>
2 {
3     T Eval(ArrayFormula cell, Sheet sheet, int col, int row);
4     T Eval(BlankCell cell, Sheet sheet, int col, int row);
5     T Eval(Formula cell, Sheet sheet, int col, int row);
6     T Eval(NumberCell cell, Sheet sheet, int col, int row);
7     T Eval(TextCell cell, Sheet sheet, int col, int row);
8 }
```

**Listing 1:** The interface for evaluating cells in Funcalc.

# 10 Implementation of Cost Semantics

Before we discuss the full implementation details of the various sections presented thus far, we give a brief introduction to some of the inner workings of the research spreadsheet application Funcalc deemed necessary for following the implementation. Readers interested in learning more are encouraged to read [1].

## 10.1 Funcalc

Besides sheet defined functions (SDFs) that are compiled to Common Intermediate Language (CIL) bytecode, ordinary cells are interpreted to values. Figure 27 shows the complete hierarchy of cells, expressions and values in Funcalc, and their relations. For example `NumberCell` and `TextCell` contain number and text constants respectively while `Formula` holds a formula expression such as `=1+2`.

Funcal's interpreter implements two interfaces for evaluating cells (`ICellEvaluator`) and expressions (`IExpressionEvaluator`), the former interface is given in Listing 1 with some details omitted. Each evaluation of a cell type happens in the context of some column and row in some sheet in the spreadsheet. These interfaces can be implemented by any class that needs to operate on cells, expressions or both, and we use them to implement our cost semantic rules.

The interpreter returns a subclass of the `Value` class used to represent all values as shown in Figure 27. For example, the `ArrayValue` holds a first-class array value. In the case of a cost interpreter, we instead want to return both a value and an associated cost so we define a cost result `CostResult` tuple which contains both elements and is the return value of our implementations. We define auxiliary some functions like `MakeCostResult` which constructs a `CostResult` tuple from a pre-existing `CostResult` or from a value and a cost.

We have implemented two variants of the cost semantics in Funcalc: a concrete cost evaluator (Section 10.2) and an abstract cost interpreter (Section 10.5). The former uses unit costs and is not guaranteed to terminate. We also discuss a few important details regarding proper handling of early- and late-bound arguments as well as cost evaluation of SDFs. The latter is inspired by [3, 10, 11].

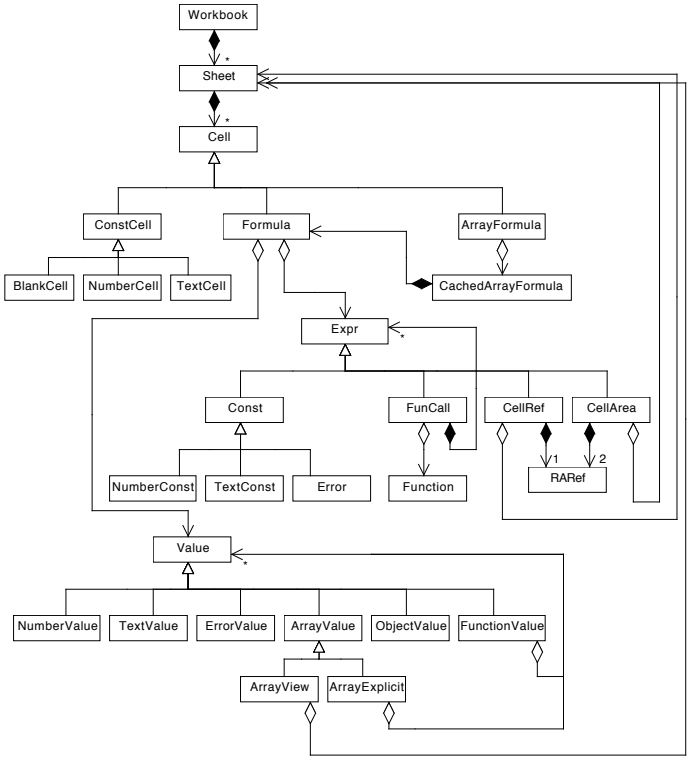


**Figure 27:** Funcalc's class hierarchy from [1]. A triangular arrow denotes inheritance with the arrow pointing at the base class; an arrow originating in an open rhombus denotes aggregation; and an arrow originating in a solid rhombus denotes composition. A number or a star at an arrowhead denotes many-to-one relations. For example, a single-cell reference `CellRef` contains a single relative or absolute cell reference `RARef`.

```
1  private CostResult EvalIf(FunCall expr, Sheet sheet, int col, int row)
2  {
3      if (expr.expressions.Length != 3) {
4          return MakeUnitCost(ErrorValue.argCountError);
5      }
6
7      CostResult condition = expr.es[0].Eval(this, sheet, col, row);
8
9      if (condition.Value is ErrorValue ev) {
10         return MakeCostResult(condition);
11     } else {
12         NumberValue n0 = condition.Value as NumberValue;
13
14         if (n0 != null) {
15             int index = n0.value != 0.0 ? 1 : 2;
16             CostResult result = expr.expressions[index].Eval(this, sheet,
                   col, row);
17
18             return MakeCostResult(result.Value, condition.Cost +
                   result.Cost);
19         } else {
20             return MakeCostResult(ErrorValue.argTypeError, condition.Cost);
21         }
22     }
23 }
```

**Listing 2:** Simplified C# code for the cost evaluation of `IF`

## 10.2 Cost Evaluator Implementation

The implementation of the cost evaluator follows the semantic cost rules closely as shown in Listing 2 for the simplified implementation of the cost evaluation of `IF` (see rules $(c3e)$, $(c3f)$ and $(c3t)$ in Section 4.1). The evaluation function `EvalIf` takes the function call expression `FunCall` representing the `IF` expression and the column, row and sheet of the cell. First, we check if the function call consists of three sub-expressions (a condition and two branches). If not, we return an error indicating an incorrect number of arguments. Otherwise, we evaluate the conditional expression using the cost evaluator. If the result is an error value, we short-circuit as per rule $(c3e)$ and return the result of the condition (the error) and the cost obtained so far. Otherwise, we cast the result of the condition to a number. If the cast fails, we return an error indicating an argument type error and the cost obtained thus far. If the condition is indeed a number, we pick the appropriate branch and evaluate the expression as per rules $(c3f)$ or $(c3t)$, then return its value along with the cost of evaluating the condition and the given branch expression. As an example, `EvalIf` would return a cost result consisting of the value `SIN(1+2)`$\approxeq 0.14112$ at cost 6 for the following expression.

```
=IF(1, SIN(1+2), COS(3))
```

Evaluation of the `IF` function call and its condition costs 2. The inner function call to `SIN` costs four: one for the `SIN` function application, one for the `+` operator application and one for each of the arguments of the addition.

```
1  void MergeArgs(Value[] early, Expr[] late, Expr[] merged)
2  {
3      int j = 0;
4
5      for (int i = 0; i < early.Length; i++) {
6          if (early[i] != ErrorValue.naError) {
7              // Wrap the value as a constant expression
8              merged[i] = Const.Make(early[i]);
9          } else {
10             merged[i] = late[j++];
11         }
12     }
13 }
```

**Listing 3:** Merging early- and late-bound arguments in the cost evaluator.

## 10.3 Early and Late Argument Binding

Our cost evaluator, and the abstract interpreter we discuss later, both operate primarily on expressions which warrants some extra considerations e.g. when we implement partial evaluation and early- and late-bound arguments. Funcalc supports partial evaluation of functions via the `CLOSURE` intrinsic function. For example, the following expression creates a closure that takes a single argument and adds one to it. Funcalc uses the `NA` function, that returns the "not available" error `#N/A!`, to denote late-bound arguments.

```
=CLOSURE("+", 1, NA())
```

In Standard ML, the equivalent expression would be `fn arg => 1 + arg`. Upon applying the closure with its remaining arguments, both the early-bound argument (`1`) and the late-bound argument (`2`) are passed together to the function call, in this case the addition operator.

```
=APPLY(CLOSURE("+", 1, NA()), 2)
```

As the cost evaluator operates primarily on expressions, we need to handle a mix of early- and late-bound values and expressions and ensure that all arguments get passed along to the final function call. Consider the more complex case of evaluating the cost of the following higher-order function call which applies a function with two early-bound arguments to each cell in the cell area `A1:A50`.

```
=MAP(CLOSURE("SUM", 1, 2, NA()), A1:A50)
```

Upon each application of the closure, we must merge the early-bound argument values `1` and `2` with the late-bound expression of the placeholder argument denoted by `NA()` and pass them to the `SUM` function call as expressions. The process is shown in Listing 3 where all arguments are merged in a single array of expressions (denoted by the Funcalc `Expr` class). Early arguments are wrapped as constant expressions using a call to `Const.Make`.

| | A | B |
|---|---|---|
| **1** | `=DEFINE("factorial", B3, B2)` | |
| **2** | `'n=` | 0 |
| **3** | `'out=` | `=IF(B2<=0, 1, B2*FACTORIAL(B2-1))` |

**Figure 28:** A recursive factorial SDF function.

## 10.4 Evaluation of Sheet-Defined Functions

In the original implementation of Funcalc sheet-defined functions are not interpreted but automatically compiled to CIL bytecode [1]. Therefore, we cannot use the existing interpreter framework directly to interpret the bytecode to find the execution cost. We could generate additional code to compute costs but this seems like an excessive and complicated approach. Instead, we directly interpret the cells of an SDF using the cost evaluator by evaluating the output cell of an SDF and follow dependencies back to its input cells. This requires proper abstraction of $\rho : Addr \to Value$, the local cell environment or stack frame of an SDF as described in Section 3.4, in order to handle both recursive SDFs and normal function calls. Consider the definition of the factorial function in Figure 28.

To implement $\rho$, we could directly modify the input cells of the SDF on each call but this would temporarily modify cells in the spreadsheet which could easily lead to inconsistencies if we are not careful. We have chosen instead to keep track of an internal, local environment $lenv : Addr \to Value$ that mimics $\rho$. When an SDF is called, we create and push a new local environment onto an internal stack and store the SDF's parameters in it by mapping the addresses of the input cells to their respective parameter values. This mimics the semantic rule for application (see ($g8$)) where the input parameters for the current function call are stored in $\rho'$ i.e. $\rho'(in_1) = v_1 \ldots \rho'(in_n) = v_n$. Upon invoking a recursive function call, we create and push a new local environment with the new parameters. When the recursive call returns, we pop the top-most local environment from the stack. Therefore, $lenv$ behaves exactly like a stack frame following the intuition in Section 3.5. We still need one last detail for the local environment to work. Cost evaluation of a cell reference is modified to first look in the top-most local environment, if any, before examining the cells of the actual sheets. Thus when we do computation in some recursive SDF and need to evaluate an input parameter, we first look in the local environment and not the actual spreadsheet.

Interestingly, if we were to strip away any notion of cost from the evaluation of SDFs, we have in fact implemented a full-fledged SDF interpreter which is likely what Funcalc would have used if there was no SDF compiler. Note that one problem with the above described approach is that it might be the case that the cost of interpretation and the cost of bytecode execution may not correlate. This is not a problem as long as cost is only used as a measure of computational

steps. However, if we were interested in worst case execution times the tighter correspondence with the CIL bytecode becomes paramount.

## 10.5 Abstract Cost Evaluator Implementation

This section presents examples from the abstract cost-semantics implementation. The abstract-cost implementation introduces a new type of value, `Top`, which represents unknown values, such as input values for the spreadsheet, and values that through computation depend on a Top value. Essentially, this is just a subclass `Top` of `Value`; a function `Top` is introduced to produce a top value. In the abstract cost-evaluator implementation we consider Array($\top$,$\top$) a Top-value.

The abstract-cost implementation is an implementation of the *evaluator* interfaces, and is essentially a modified version of the `CostEvaluator`. Specifically, the difference is special handling of some expressions.

Such expressions are:

- branching expressions, such as if `IF`, explained in Listing 4. The implementation of other branching expressions, such as `And`, `Or`, `CountIf` etc. are modified as expected.
- `Closure` and `Apply`, where the result is `Top` if the first argument is `Top`, and evaluated as the `CostEvaluator` otherwise.
- The `Map`-family of functions, `HScan` and `VScan`, and `Tabulate`, all result in the value `Top` with cost $\infty$, in case any arguments are `Top`.
- Function calls, where the result is top with the cost of evaluating the arguments plus the cost of evaluating the function, in case any of the arguments are top. The result is always top, as a top argument may be error.

To handle top values, the else-branch in line 11 of Listing 2 is modified as shown in Listing 4.

In this modification, if the condition is Top, the result is also Top, with cost of the expensive branch with an added cost of the condition-evaluation cost plus the *unitcost* of the if-expression. Otherwise, the result is the result of evaluation by the `CostEvaluator`-implementation.

Taking the previous example from Section 10.2 with a top value instead of a numeric value as condition:

```
=IF(Unknown(), SIN(1+2), COS(3))
=IF(Unknown(), SIN(3), COS(1+2))
```

the result of both the above expressions is `Top` with the cost of 6.

```
1  public Value EvalIf(FunCall expr, Sheet sheet, int col, int row)
2  {
3      CostResult condition = expr.es[0].Eval(this, sheet, col, row);
4
5      // ...
6
7      if (condition.Value is NumberValue n0) {
8          int index = n0.value != 0.0 ? 1 : 2;
9          CostResult result = expr.expressions[index].Eval(this, sheet, col,
               row);
10         return MakeCostResult(result.Value, condition.Cost + result.Cost);
11     } else if (condition.Value is Top) {
12         CostResult tt = expr.es[1].Eval(this, sheet, col, row);
13         CostResult ff = expr.es[2].Eval(this, sheet, col, row);
14         var cost = (tt.Cost > ff.Cost ? tt.Cost : ff.Cost) + condition.Cost;
15         return MakeCostResult(new Top(), cost);
16     } else {
17         return MakeCostResult(ErrorValue.argTypeError, condition.Cost);
18     }
19 }
```

**Listing 4:** Simplified C# code for abstract cost-evaluation of `IF`

# 11 Results

In this section, we present our results for the concrete and abstract cost evaluators.

## 11.1 Concrete Cost Evaluator Results

Since the concrete cost evaluator costs are proportional to the number of operations of an expression or alternatively the number of rule applications, we are not particularly interested in the precision of the costs. Instead, we are interested in how long it takes to evaluate the cost of each cell in a spreadsheet.

Table 1 contains the concrete and abstract costs, number of formula cells and time taken to evaluate the cost of all cells in six spreadsheets from LibreOffice Calc [12] and a subset of the EUSES corpus [13]. The costs correspond to applying the $\gamma$ function to each cell address $ca$ in the spreadsheet as presented in Section 5.3. Similarly, the running time is the time taken to evaluate the cost of each cell in the spreadsheet. Note that running times for the LibreOffice Calc spreadsheets are given in seconds while the running times for the EUSES spreadsheets are given in milliseconds. The third and fourth columns relate to abstract costs and are discussed in the next section.

## 11.2 Abstract Cost Evaluator Results

The abstract cost evaluator would in a standard spreadsheet compute the same values as the Concrete Cost Evaluator, and the same costs, since there are no

| Spreadsheet | Total Concrete Cost | Abstract Cost | Inc. | Formulas | Runtime |
|---|---|---|---|---|---|
| LibreOffice Calc (runtime in *seconds*) | | | | | |
| building-design | 978 520 000 | 978 520 000 | 100% | 108 332 | 33.64 |
| energy-markets | 2 175 001 469 | 2 175 001 469 | 100% | 534 507 | 3011.96 |
| grossprofit | 4 423 203 701 | 4 423 203 701 | 100% | 135 073 | 2324.62 |
| ground-water | 1 099 998 389 | 1 099 998 389 | 100% | 126 404 | 79.39 |
| stock-history | 1 230 276 358 | 1 230 276 358 | 100% | 226 503 | 85.30 |
| stocks-price | 1 165 235 199 | 1 165 235 199 | 100% | 812 693 | 1344.60 |
| EUSES (runtime in *milliseconds*) | | | | | |
| 2004_PUBLIC_BUGS_INVENTORY | 140 925 | 140925 | 100% | 4495 | 28.83 |
| Aggregate20Governanc#A8A51 | 723 436 | $\infty$ | NA | 3546 | 154.93 |
| high_2003_belg | 11 616 516 | $\infty$ | NA | 12 861 | 58.56 |
| DNA | 127 029 | 127029 | 100% | 4715 | 15.76 |
| EUSE | 3463 | 3463 | 100% | 413 | 1.27 |
| PLANCK | 25 200 | 25200 | 100% | 806 | 13.33 |
| 02rise | 91 581 | 91581 | 100% | 10 316 | 26.64 |
| financial-model-spreadsheet | 20 128 | $\infty$ | NA | 3115 | 10.99 |
| Financial-Projections | 31 400 | 31994 | 101.9% | 3649 | 11.04 |
| 2000_places_School | 9286 | 9286 | 100% | 1375 | 2.39 |
| 2002Qvols | 10 222 | 10222 | 100% | 2184 | 2.35 |
| EducAge25 | 34 058 | 34058 | 100% | 1470 | 6.19 |
| notes5CMISB200SP04H2KEY | 156 093 | $\infty$ | NA | 1557 | 103.60 |
| Test20Station20Powe#A90F3 | 15 720 | 15720 | 100% | 2164 | 5.59 |
| v1tmp | 6157 | 6257 | 101.62% | 1129 | 2.06 |
| MRP_Excel | 415 529 | $\infty$ | NA | 4809 | 92.16 |
| ny_emit99 | 76 010 | 76010 | 100% | 4352 | 24.28 |
| Time | 33 832 | 33832 | 100% | 4198 | 6.65 |
| WasteCalendarCalculat#A843B | 10 309 | 11901 | 115.44% | 843 | 1.81 |
| funding | 280 702 | $\infty$ | NA | 1636 | 215.05 |
| iste-cs-2003-modeling-sim | 14 919 | 14919 | 100% | 1991 | 6.71 |
| modeling-3 | 1292 | 1292 | 100% | 213 | 0.54 |

**Table 1:** The total concrete cost, the abstract cost, overapproximation in the abstract cost evaluation, number of formula cells and the time taken to evaluate the cost of all cells in the LibreOffice Calc and EUSES spreadsheets. The cost evaluation was run twenty times and the averages of those runs are shown in the fourth column.

values resulting in an abstact calculation, i.e. a *Top* value. In our evaluation of the abstract cost, we replace all constants in the spreadsheet by `Top` values, before the abstract cost evaluator is run. The results are found in table 1. Because some top values are used in conditions in recursive calls, some cost-results are $\infty$, caused by infinite recursion. The largest overapproximation is found in `WasteCalendarCalculat#A843B`. This is caused by a large number of `IF`s, of the form: `IF(T11>-1, 0, IF(T11<1, (S11-F11)/F11, 0))` where `T11` is a top value in the abstract version. Other spreadsheets have either cost-balanced branchings or the most expensive branch is also taken in concrete evaluation.

## 11.3 Discussion

At a glance, we notice that there seems to be no correlation between the number of formula cells and the time taken to evaluate the cost of each cell in the spreadsheet. This is to be expected as the formula count does not tell us anything about the complexity about each individual formula. For example, the

`ny_emit99` and `Time` spreadsheets have almost the same number of formula cells but vastly different concrete and abstract costs and runtime.

# 12 Conclusion and Future Work

The evaluation semantics for simple Funcalc expressions was elaborated in Section 2 and semantics for extended spreadsheet expressions was developed in Section 3. In section crefsec-cost-semantics a precise cost semantics built on top of the extended semantics was presented in Section 5 and in Section 6. This cost semantics serves as a foundation for the concrete and abstract cost calculation functions described in Section 7. The extended evaluation semantics for Funcalc was extended to compute with unknown values in Section 8, which serves as a first step towards an approximate cost analysis described in Section 9. Implementations for the concrete and abstract cost semantics were presented in Section 10. Finally in Section 11,we presented results pertaining to the execution time and precision of the various cost analyses that were implemented as described in Section 10.

The purpose of the cost semantics and calculations is to serve as a guide for load-balancing parallel computations in spreadsheets, e.g. via task partitioning for execution on multi-core CPUs [7] or off-loading work to GPGPUs [14]. Moreover, the evaluation and cost semantics may serve to improve the understanding of spreadsheet computations in general and the safety of and reliance on a given implementation. Also, they may be used to prove that optimizations preserve the meaning of spreadsheet computation and that these optimizations reduce the amount of work needed to perform a computation.

The approximate cost analysis is a first step towards a more general framework of abstract interpretation of spreadsheet expressions, based on ideas presented in [3]. Due to the higher-order nature of Funcalc, another future development would be a closure analysis to improve cost estimates of function application.

One future development may be to give a precise semantics for *depth* (also called *span* or *critical path length*), the length of the longest sequential dependence, in the sense of Blelloch [2], for parallel evaluation. This could be used as basis for an abstract interpretation to estimate *depth* in addition to the *work* defined in this paper.

Finally, one could also imagine various tools, based in the formal semantics, for analyzing or verifying various aspects of spreadsheets. One such tool could be a tool to formally verify the correctness of the spreadsheet program. Another tool could guide users through performance bottlenecks in a spreadsheet and even suggest possible improvements.

# References


[1] Peter Sestoft. *Spreadsheet Implementation Technology*. The MIT Press, 2014. ISBN: 9780262526647.

[2] Guy E. Blelloch. "Programming Parallel Algorithms". In: *Commun. ACM* 39.3 (Mar. 1996), pp. 85–97. ISSN: 0001-0782. DOI: 10.1145/227234.227246. URL: http://doi.acm.org/10.1145/227234.227246.

[3] David A. Schmidt. "Trace-Based Abstract Interpretation of Operational Semantics". In: *LISP and Symbolic Computation* 10.3 (May 1998), pp. 237–271. ISSN: 1573-0557. DOI: 10.1023/A:1007734417713. URL: https://doi.org/10.1023/A:1007734417713.

[4] Gustavo Gómez and Yanhong A. Liu. "Automatic Time-bound Analysis for a Higher-order Language". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '02. Portland, Oregon: ACM, 2002, pp. 75–86. ISBN: 1-58113-455-X. DOI: 10.1145/503032.503039. URL: http://doi.acm.org/10.1145/503032.503039.

[5] Mads Rosendahl. "Automatic Complexity Analysis". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: ACM, 1989, pp. 144–156. ISBN: 0-89791-328-0. DOI: 10.1145/99370.99381. URL: http://doi.acm.org/10.1145/99370.99381.

[6] Chris Scaffidi. *Counts and earnings of end-user developers*. Sept. 21, 2017. URL: https://www.linkedin.com/pulse/counts-earnings-end-user-developers-chris-scaffidi?published=t (visited on 10/19/2017).

[7] Thomas Bøgholm et al. "Analyzing spreadsheets for parallel execution via model checking". In: *Essays on the Occasion of Bernhard Steffen's 60th Birthday (Lecture Notes in Computer Science, vol. 11200)*. Springer-Verlag, 2018.

[8] Gilles Kahn. "Natural Semantics". In: *STACS '87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany (Lecture Notes in Computer Science, vol. 247)*. Ed. by F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing. Springer-Verlag, 1987, pp. 22–39.

[9] H.R. Nielson and F. Nielson. *Semantics with Applications. An Appetizer*. Springer-Verlag, 2007.

[10] David Darais et al. "Abstracting Definitional Interpreters". In: *CoRR* abs/1707.04755 (2017). arXiv: 1707.04755. URL: http://arxiv.org/abs/1707.04755.

[11] Daniel P. Friedman and Anurag Mendhekar. *Tutorial: Using an Abstracted Interpeter to Understand Abstract Interpretation*. Tech. rep. Course notes for CSCI B621. Indiana University, 2003.

[12] The Document Foundation. *LibreOffice Calc*. URL: https://www.libreoffice.org/discover/calc/ (visited on 11/13/2018).

[13] Marc Fisher and Gregg Rothermel. "The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms". In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1082983.1083242. URL: http://doi.acm.org/10.1145/1082983.1083242.

[14] Jim Trudeau. *Collaboration and Open Source at AMD: LibreOffice*. Accessed on 31.07.2015. July 2015. URL: https://developer.amd.com/collaboration-and-open-source-at-amd-libreoffice/.

# A Table of Transformation Functions

| Transformation | Description | Category |
|---|---|---|
| $T_v$ | Transforms expressions to their corresponding values, constructing time-value pairs for closure expressions. | Concrete |
| $T_t$ | Transforms expressions to their corresponding costs. | Concrete |
| $T_d$ | Transforms *sdf* to their corresponding time functions $sdf_t$, obviating the need for a time-value pair for closure expressions. | Concrete |
| $T_{vb}$ | Transforms expressions to their corresponding abstract values accounting for unknown values. | Abstract |
| $T_{at}$ | Abstract time function that transforms expressions to their corresponding abstract costs. | Abstract |